Parallel Incremental Compilation

Neal M Gafter

Technical Report 349
June 1990

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Parallel Incremental Compilation

by

Neal M Gafter

Submitted in Partial Fulfillment

of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

Supervised by Thomas J. LeBlanc

Department of Computer Science

University of Rochester

Rochester, New York

June 1990

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 349	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Parallel Incremental Compilation	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Neal M. Gafter	8. CONTRACT OR GRANT NUMBER(s): N00014-82-K-0193
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Science Department 734 Computer Studies Bldg. University of Rochester, Rochester, NY 14627	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Project Agency 1400 Wilson Blvd. Arlington, VA 22209	12. REPORT DATE: June 1990 13. NUMBER OF PAGES: 113
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research Information Systems Arlington, VA 22217	15. SECURITY CLASS. (of this report): Unclassified. 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution for this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

parallel incremental compilation; parallel algorithm complexity; parsing; semantic analysis; upward remote aggregate attribute grammars

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

20. ABSTRACT

The time it takes to compile a large program has been a bottleneck in the software development process. When an interactive programming environment with an incremental compiler is used, compilation speed becomes even more important, but existing incremental compilers are very slow for some types of program changes. We describe a set of techniques that enable incremental compilation to exploit fine-grained concurrency in a shared-memory multiprocessor and achieve asymptotic improvement over sequential algorithms. Because parallel non-incremental compilation is a special case of parallel incremental compilation, the design of a parallel compiler is a corollary of our result.

Instead of running the individual phases concurrently, our design specifies compiler phases that are mutually sequential. However, each phase is designed to exploit fine-grained parallelism. By allowing each phase to present its output as a complete structure rather than as a stream of data, we can apply techniques such as parallel prefix and parallel divide-and-conquer, and we can construct applicative data structures to achieve sublinear execution time. Parallel algorithms for each phase of a compiler are presented to demonstrate that a complete incremental compiler can achieve execution time that is asymptotically less than sequential algorithms.

Historically, the symbol table has been a bottleneck to parallel compilation; no previously described algorithm executes in time less than linear in the number of declarations. We describe new algorithms for parsing using a balanced list representation and type checking based upon attribute grammars modified with a combination of aggregate values and upward remote references. Under some mild assumptions about the language and target program, these phases run in polylogarithmic time using a sublinear number of processors.

The design of computer languages has been influenced by the compiler technology available; we show how some language design decisions can simplify the design of a parallel incremental compiler, allowing more efficient algorithms to be used.

Curriculum Vitae

Neal M Gafter was born in Buffalo, New York on January 11, 1960. He took an academic interest in software during his Junior and Senior years at Sweet Home Senior High School, and began his professional career as a developer of scientific software for Sierra Research Corporation (now LTV Aerospace and Defense) during summers and academic breaks after High School, starting in 1977 and throughout his undergraduate years. Neal studied Computer Engineering at Case-Western Reserve University, where he was a recipient of various academic scholarships and appeared frequently on the Dean's list. His first publication, submitted as a senior project, was a technical report for Sierra entitled "Optimal Multitheodolite Position Estimate." His studies at CWRU led him to take a strong interest in systems software, particularly compilers and operating systems. He graduated from CWRU in May of 1981 with high honors.

In June of 1981 Neal took a job with a newly formed group at Xerox in Rochester, New York, whose charter was to develop a language and software development environment for distributed embedded real-time systems. At the same time, Neal began half-time study at the University of Rochester, leading to a Master's degree in May of 1984. In January of 1985, he left Xerox and began full-time study at the University of Rochester.

At the University of Rochester, under his advisor Tom LeBlanc, Neal studied languages and paradigms for distributed and parallel programming. One of his first projects

was the implementation of a Modula-2 compiler and runtime system for the BBN Butterfly parallel processor. He was a co-author and implementor for "SMP: A Structured Message-Passing System for the BBN Butterfly." He also was a co-author and implementor for the DARPA benchmark project and paper "Parallel Computational Geometry." In cooperation with an operating systems seminar, he was co-author of "The Elmwood Multiprocessor Operating System," which appeared in *Software — Practice and Experience*, November 1989. At the 1987 International Conference on Parallel Processing, he presented "Algorithms and Data Structures for Parallel Incremental Parsing," out of which this dissertation grew. He has also taken part in the implementation of the Psyche multiprocessor operating system.

Acknowledgments

I would like to thank, first and foremost, my wife Ricki for her patience and support during the years, and particularly the final months of my study.

Tom LeBlanc was my research advisor for this dissertation. Tom has a knack of giving just the right amount and kind of direction, neither more nor less than necessary.

I would like to thank the Faculty and staff of the Department of Computer Science at the University of Rochester for creating an ideal research environment, in which the graduate students are able to pursue their own research interests.

I would especially like to thank the members of the Elmwood group for many enjoyable discussions and debates.

This material is based upon work supported by the National Science Foundation under Contract number CCR-8320136. The Government has certain rights in this material. This work was also supported by ONR/DARPA Research Contract number N00014-82-K-0193.

Abstract

The time it takes to compile a large program has been a bottleneck in the software development process. When an interactive programming environment with an incremental compiler is used, compilation speed becomes even more important, but existing incremental compilers are very slow for some types of program changes. We describe a set of techniques that enable incremental compilation to exploit fine-grained concurrency in a shared-memory multiprocessor and achieve asymptotic improvement over sequential algorithms. Because parallel non-incremental compilation is a special case of parallel incremental compilation, the design of a parallel compiler is a corollary of our result.

Instead of running the individual phases concurrently, our design specifies compiler phases that are mutually sequential. However, each phase is designed to exploit fine-grained parallelism. By allowing each phase to present its output as a complete structure rather than as a stream of data, we can apply techniques such as parallel prefix and parallel divide-and-conquer, and we can construct applicative data structures to achieve sublinear execution time. Parallel algorithms for each phase of a compiler are presented to demonstrate that a complete incremental compiler can achieve execution time that is asymptotically less than sequential algorithms.

Historically, the symbol table has been a bottleneck to parallel compilation; no previously described algorithm executes in time less than linear in the number of declarations.

We describe new algorithms for parsing using a balanced list representation and type checking based upon attribute grammars modified with a combination of aggregate values and upward remote references. Under some mild assumptions about the language and target program, these phases run in polylogarithmic time using a sublinear number of processors.

The design of computer languages has been influenced by the compiler technology available; we show how some language design decisions can simplify the design of a parallel incremental compiler, allowing more efficient algorithms to be used.

Table of Contents

List of Figures

1 Introduction

The modern approach to software development environments has all of the tools working together as one integrated package [Delisle *et al.*, 1984]. The user, concerned only with editing, running, and debugging the program, need not be aware of the existence of a compiler. Interpreted systems are often closer to this ideal, are typically more integrated, and require less support from the user than compiled systems. It is an accident of history that users explicitly invoke compilers and linkers before running their programs.

Incremental compilers can present the same user model for program development as interpreters while executing user programs significantly faster [Feiler, 1982; Fritzson, 1983; Schwartz *et al.*, 1984; Fritzson, 1984]. An environment implemented with an incremental compiler enables the user to change the program as it is being run and debugged; ideally the program is ready to be run immediately after the changes are complete. Unfortunately, few existing incremental compilers are fast enough to suit these needs. It is the goal of this thesis to attack this problem by exploiting fine-grained parallelism in incremental compiler design, with asymptotic complexity as a measure of our success.

Complexity theory gives us important notions of algorithm efficiency that have been fruitful in sequential program design; similarly, parallel complexity theory helps us to understand parallel program efficiency. With the advent of larger and more powerful multiprocessors, algorithms that have a time complexity that improves with increasing

numbers of processors are becoming more important for their effective use. Unfortunately, most recent work on parallel compilation has focused on the design, implementation, and performance evaluation of compilers for specific machines with a small number of processors without an asymptotic analysis of the complexity of the underlying algorithms.

For example, parallel compilers have been constructed using a pipeline strategy, running the various phases of the compiler simultaneously. This technique yields only a fixed speedup limited by the number of compiler phases. Although these studies help us to evaluate the constant factors in an implementation, this should be secondary to a design and analysis based on asymptotic complexity. Since sequential compilation has linear time complexity, the goal of parallel compilation should be *sublinear* time complexity: speedup that improves as the size of the program grows.

We describe a design that, by exploiting fine-grained, data-level parallelism within each phase of the compiler, can achieve sublinear execution time with a sublinear number of processors. For example, under some mild assumptions about the distribution of target programs, our algorithm for parsing runs in time $O(\log^3 n)$ using $O(n/log^3 n)$ processors, and our type checking algorithms run in $O(\log^3 n)$ time using $O(n/\log n)$ processors; fewer processors are needed when only small changes have been made to the program.

Rather than present the design of a single compiler in detail, we present a set of techniques that can be used to construct tools for building parallel incremental compilers for different languages. Therefore, we rely on techniques that are suitable for describing compilers in formal terms. The two main description languages are extended context free grammars extended and extended attribute grammars. We also discuss how to use these techniques in describing a parallel incremental compiler.

This thesis is also an extended example of parallel program design for a familiar but realistically complex problem. We identify and discuss a number of parallel program design techniques and principles that we have used throughout the compiler.

1.1 The Incremental Programming Environment

We believe the usual model of structured editing imposed by incremental compilers is too restrictive [Waters, 1982]. Incremental compilers have used structured editors in part because it is much easier to integrate an editor with an incremental compiler when the editor is language-specific. We show how an incremental compiler can be integrated with any text editor that has been modified to output a log of editing actions. Thus, the user can apply a familiar editing paradigm while reaping the benefits of an integrated environment.

The compiler described by our design reads this log as well as portions of the source file to begin recompilation. Because the old version of the parse tree and symbol table are maintained, the compiler need only examine the changes that appear in the source file. Thus, the amount of work to recompile a program is related to the scale of the changes made to it. By distributing the workload over a number of parallel processors, we can reduce the elapsed time to complete compilation.

The structure of the compiler is based on a conventional syntax-directed translation schema, with the individual phases of compilation storing their intermediate results in a form that can be updated. In addition, the algorithms used in the design of each phase exploit fine-grained parallelism. Because a compiler is limited in execution time by its slowest phase, we describe parallel algorithms for each phase of compilation and analyze the individual phases and aggregate complexity.

Our main contribution to the design of incremental compilers is an extension to

the attribute grammar formalism for symbol table maintenance called *upward remote aggregates*. This technique is based on a separation of the problem of symbol table and scope analysis (the question of *where* a definition is visible) from the independent problem of evaluating the identifier definitions and other semantic attributes (the question of *what* is the definition of a symbol) in an order defined by the attribute and symbol dependence graph.

1.2 Thesis Statement

It is the thesis of this dissertation that *parallel algorithms can be used to achieve an asymptotic improvement in running time over sequential algorithms for incremental compilation.*

We demonstrate the thesis by providing a set of techniques to implement an incremental compiler that exploits fine-grain parallelism in the compilation process. Formal methods are used so that the techniques developed may be used for a variety of languages. We present and analyze the complexity of our techniques relying, when necessary, on assumptions about the characteristics of the programming language, the program being compiled, and the set of program changes. The result is a design that allows incremental compilation to be performed in parallel in sublinear time.

Parallel compilation is an important special case of parallel incremental compilation that has seen some previous research. However, there has been no previous work that describes and analyzes algorithms for parallel semantic analysis and its subproblem of symbol table maintenance. We present new techniques that are amenable to analysis.

In time $O(\log^2 n)$ using $O(n)$ processors, we construct a graph of dependencies among semantic attributes, identifier definitions and identifier references. After the graph is constructed, the semantic attributes and definitions of the identifiers may be evaluated

in time dependent only on the depth of the semantic dependency graph.

The running time of the compiler as a whole is bounded by the running time of its slowest phase, and so speedup in each and every phase is important; parallelism between phases does not improve the asymptotic complexity beyond a constant factor. In order to achieve our goal of asymptotic sublinear execution time, each phase of the compiler must be highly parallel. If any part of the compiler has a linear factor in its expected running time, then the execution time of the entire compiler will be linear. In our design, each phase may complete before the next begins. Thus, the running time of the compiler as a whole is the sum of the running times of the individual phases. Because there are a constant number of phases, this is proportional to the running time of the slowest phase.

We do not hesitate to split an algorithm into a constant number of mutually sequential but highly parallel phases. By consistently applying this design approach to problems throughout the compiler we derive a compiler design in which each phase is simple enough to be amenable to analysis, enabling us to relate the complexity of the compiler directly to features of the language and target program.

1.3 Dissertation Overview

The organization of this thesis is based on the phase structure of the compiler. Our compiler design begins with a text editor that the user applies to the source text of the target program. We assume a conventional editor has been modified to output a log of editing actions, expressed as insertions and deletions (since all actions can be expressed in terms of these). When the user has completed a set of changes, the compiler is invoked, either automatically, by user request, or at periodic intervals.

Chapter 2, *Lexical and Syntactic Analysis*, presents algorithms that process the

editor log to determine what parts of the program contain new text. The parser then constructs a new parse tree by assembling pieces of the old parse tree and creating new parse tree nodes where the program has changed, invoking lexical analysis on any new text. The syntax of the language is described by the compiler writer using context free grammars extended with a new notation for lists that is used to improve the efficiency of parallel parsing by creating balanced parse trees. Our analysis shows that parsing takes $O(\log^3 m)$ time in m, the number of editing opertions or $O(\log^3 n)$ time in n, the size of the program, using $O(n/\log^3 n)$ processors. If the program is syntactically correct, the compiler continues with semantic analysis.

Chapter 3, *Parallel Separate Compilation*, is a slight departure from our exposition of the compiler design. In it we discuss the use of parallelism to process a *makefile*, and in the separate compilation units of a single program. This is one way of exploiting parallelism at a large grain for the incremental compilation of multi-unit programs, but the chapter's main purpose is as an example of the application of techniques from Chapter 2, as well as an introduction to the techniques described in semantic analysis for the processing of the Makefile language. This chapter develops algorithms for comparing lists of items represented as balanced binary trees, to discover differences between the set of items at their leaves.

Chapter 4, *Code Generation and Semantic Analysis*, describes a new extension to the attribute grammar formalism, *upward remote aggregates*, that is used by the compiler writer to describe the scoping structure of programming languages. The formalism improves the clarity of the language description over conventional attribute grammars and provides a natural way to express scoping constructs. We describe and analyze an implementation strategy that exploits fine-grained parallelism in the evaluation of these grammars.

Semantic analysis begins by comparing the old and new parse trees using algo-

rithms developed in Chapter 3, and updates the aggregate attributes (symbol tables) to reflect any changes in scope structure of the target program. Then, the parse trees are compared once more to detect changes in the set of identifier references. After these two phases, all dependency links involving attributes in the aggregates have been constructed. A transitive closure from the set of changed attributes is used to mark all attributes affected by the changes, after which the attribute dependency graph is scheduled for evaluation using a scheduling algorithm based on a variant of the topological sort algorithm.

We discuss a number of techniques that can be used by the compiler writer to describe the semantics and code generation for typical programming languages using the extended attribute grammar formalism. The list productions used in parsing create balanced parse trees of bounded depth, for which techniques such as parallel divide-and-conquer may be used to solve many of the problems arising in semantics and code generation.

Finally, Chapter 5 summarizes our results, considers them in a practical context, and describes the major remaining open problems and directions for future work.

1.4 Previous Work

This section is a brief survey of previous work on the topic of parallel compilation that does not fit well into our subdivision of the compiler phases. Each later chapter discusses previous work related to its task, but some parallel compilation researchers take different approaches. One unifying feature of all of this work is that it is limited to a constant speedup, or put another way, s linear time in the length of the program to compile. From this point of view, they all fail to meet our criteria of success.

Pipelining has been used to exploit parallelism in compilation [Baer and Ellis, 1977;

Huen *et al.*, 1977; Miller and LeBlanc, 1982; Allen, 1987] by breaking the compiler into a number of separate phases. These phases are run simultaneously on separate processors. Unfortunately, the number of phases is built into the design of the compiler, and a typical phase must examine all of the input, so a compiler with n phases cannot achieve a speedup of more than n. When implemented, these designs report at best a speedup of about 2.7.

Vandevoorde's work [1988] was an ambitious parallelization of an existing sequential C compiler. His results show satisfactory speedup, using five processors, for all but symbol table analysis. One of the important contributions of that work is a parallel programming library that automatically clusters concurrent tasks for parallel divide-and-conquer algorithms to effectively use practical multiprocessors. This points out the practicality of very-fine-grained parallel algorithms based upon the technique of divide-and-conquer.

A recent paper [Khanna *et al.*, 1990] proposes a method for parallel compilation based upon partitioning the grammar of the language into functional subgrammars. A processor is devoted to each functional subgrammar, and the processors operate in parallel. This approach, unfortunately, is limited in parallelism by the number of subgrammars the compiler writer can identify (at most the number of nonterminals in the grammar) which is a constant independent both of the number of processors available and the size of the program.

There has been little work to date on the use of parallelism in incremental compilation. Kaplan and Kaiser [1986] consider the problem of multiple programmers working on different parts of a common program, using workstations sharing a common high-speed network. Individual source files are compiled sequentially.

1.5 Model and Assumptions

Our compiler design assumes a collection of identical shared-memory processors with uniform memory access time. Where we explicitly share data, we assume that reads are concurrent and writes are exclusive. In addition, we utilize *semaphores* without considering their implementation, and we use *concurrent search structures*, for which we describe one possible implementation.

The grain size appropriate to the target multiprocessor is of little importance to our design. Our algorithms are designed to exploit parallelism at a very fine grain, but a number of techniques are known to efficiently bundle grains in a way suitable for specific host architectures.

Finally, garbage collection would be very helpful in an actual implementation based upon the ideas described here, because of our use of applicative data structures.

2 Lexical and Syntactic Analysis

One of the most significant stumbling blocks to efficient parallel compilation is the depth of the parse tree. Algorithms that can parse programs with arbitrarily deep trees in logarithmic time are known, based both on LL [Skillicorn and Barnard, 1988] and LR [Klein and Reif, 1988] techniques. These algorithms do not directly construct a parse tree, which is useful in a complete compiler. Even if the compiler does not explicitly construct the parse tree, but only implicitly follows it while performing its analysis, the depth of the derivation followed is a constraint on the available parallelism in the compiler that is difficult to overcome.

Deep parse trees are an impediment to efficient concurrent semantic analysis and code generation because each requires, in general, a traversal of the entire parse tree. There are only two possible causes of deep parse trees: deeply nested programs and programs with long lists.

Deeply nested programs will unavoidably create deeply nested parse trees, so it is unlikely we will find algorithms for parallel compilation whose execution time will be sublinear for such programs. We believe that deeply nested programs do not appear in practice[1]. We concentrate our attention, therefore, on programs with bounded nesting depth.

[1]We would like to say that real programs are bounded in nesting depth by $O(\log n)$ or, better yet, $O(1)$ but we have no hard evidence. However, we have never seen a program nested so deeply that the indentation was wider than a display screen. This would imply a constant bound on nesting depth.

The most common cause of linear depth parse trees is lists. Lists of items (such as statements or declarations) are represented in a grammar as a left- or right-recursive nonterminal. This type of grammar generates a parse tree whose depth is proportional to the length of the list in the target program. It would be difficult to traverse such a deep tree in parallel and compute semantics of any complexity in less than linear time.

In order to make it easier to compute the semantics in sublinear time, parsing syntactic lists should be done differently. We introduce special *list productions* into the grammar describing the language. These list productions may be used by the compiler writer in place of right- or left-recursive productions to describe lists of syntactic entities.

```
<List> ::= <List> <List>
<List> ::= <Item>
```

This grammar describes all binary parse trees for a list of items. Lists described this way can be parsed into a balanced binary tree representation of the list, instead of the right-linear or left-linear binary parse trees that would be generated by the conventional recursive techniques. A list of length n may be represented by a balanced parse tree of depth $O(\log n)$. Such balanced structures can be parsed and traversed in parallel more efficiently than the linear structures.

By using an ambiguous grammar and reducing the parse tree depth, we have the potential to improve the available parallelism in parsing, semantic analysis, and code generation.

2.1 Editing Model

Most work on incremental compilers takes the point of view that the parser should be integrated into the editor, and that the user should not be allowed to enter a syntactically

incorrect program. The language-oriented editor approach has both advantages and disadvantages compared to the text editing model. Unfortunately there has been little systematic study of the impact of this paradigm on programmer productivity. Waters [1982] gives an argument in favor of the retention of text oriented commands in a program editor.

When a language-oriented editor is used, the parse tree is maintained as the program is modified, obviating the need for parallel or incremental parsing algorithms. We describe algorithms that may be used with a conventional text editor, making both parsing techniques applicable to compiler developers. Whichever form of editing is supported, the compiler can be run as a background process during editing to give the user periodic feedback.

2.2 Previous Work

A number of reserchers have looked at the problems of parallel parsing and scanning and incremental parsing and scanning, but we are the first to consider parallel algorithms for incremental parsing and scanning.

2.2.1 Lexical Analysis

Lexical analysis translates portions of the input program, which is a stream of characters, into a stream of syntactically meaningful entities, or *tokens.*

Lexical analysis can be broken down into two distinct tasks: *scanning* and *screening.* Scanning breaks up the input stream into substrings, based on a grammar describing the language's lexical structure. Scanning also produces a crude classification of the substrings based upon their structure. When implemented as a finite-state machine (FSM), this crude classification results from the identity of the final accepting state for each token.

Screening refines the classification produced by the scanner, discards blanks and comments, and distinguishes keywords from identifiers. Screening also produces a *name table* that contains the name of each identifier appearing in the input, and replaces the identifier with its index into the name table, thus producing a unique value for each distinct identifier name encountered.

The screener produces a sequence of tokens, each represented by its *token type* and an internal representation. For identifiers, the internal representation is an index into the name table; for constants, it is the value.

Parallelism in scanning may be obtained by dividing the source program into a number of segments of approximately equal size (or difficulty) and performing lexical analysis on each segment simultaneously using a single processor for each. The scanning process simulates a sequential scan by having the local processors simulate the state transitions of a sequential scanner. The difficulty of the scanning process is in determining what state of the FSM the local scanner should use at the beginning of its text segment.

This is a special case of the Parallel Prefix problem [Ladner and Fischer, 1980] that Schell [1979] has solved in $O(\log n)$ time (using $O(n)$ processors. His processor bound may be easily reduced to $O(n/\log n)$ without affecting the asymptotic running time.

Another technique is to perform a minimal amount of parsing during scanning, possibly using source-language specific techniques [Low, 1988; Seshadri *et al.*, 1987]. This parsing matches multi-line tokens across processors, handles nested comments, and determines regions of the program that might be processed independently by later phases of the compiler.

Screening is a subtask of lexical analysis that creates a single name table for the identifiers appearing in the program. A centralized name table introduces a bottleneck for screening, so it must be distributed. Schell [1979] recommends a hierarchical table

in which each processor maintains a cache of the global name table. Only when the local cache fails does the processor consult the global table. Unfortunately, the global table remains a centralized bottleneck to screening.

Jim Low [1988] solved this problem by putting one or more buckets of the centralized hash table on each processor. In this way, the memory accesses are distributed. Somewhat surprisingly, with this improvement caches of remote name table information did not improve the performance for the BBN Butterfly implementation (a shared-memory machine), because the overhead of a software-implemented cache was more than the sum of the remote memory referencing overhead plus the relatively small contention on the infrequent hash-table collisions. Low's results provide evidence that, given a sufficient hash table size, scanning can be performed in nearly constant time for a wide range of source file lengths.

Yu [Yu, 1989] has recently published a survey of parallel algorithms for lexical analysis. The completeness of that work allows us to treat parallel lexical analysis as a solved problem, so we consider it only briefly.

A number of researchers have considered the problem of concurrent search structures, for which the screener's identifier table is a single application [Ellis, 1985; Shasha and Goodman, 1988]. We describe in Section 2.4.3 a concurrent extensible has table suitable for the screener.

2.2.2 Incremental Parsing

Morris and Schwartz [1981] describe a structured editor based on incremental parsing. A "split" of the parse tree is maintained at the current position of the cursor. Although the algorithm supports editing using the textual viewpoint, it requires the use of a special editor, and it must consider the user's changes sequentially as they are entered.

Ghezzi and Mandriolli [1979] describe a sequential incremental parsing algorithm for

a single change to the text file. Parsing begins just to the left of a single change, and to avoid parsing the rest of the program, a similar reparse from right to left begins just to the right of the change.

Jalili and Gallier [1982] describe a general algorithm for sequential incremental parsing based on a number of changes to the text file. The algorithm assumes the existence of an easily computable function from parse nodes to source positions and the reverse; their implementation provides this by integrating the parser (and the parse tree data structure) into the editor. Stromberg [1982] separates the implementations of text editor and parser by having the editor output a log of the users editing actions. Attributes in the parse tree record the length of the text derived from each nonterminal, allowing easy location of affected nodes. These attributes are unaffected by nonlocal changes.

Poe [Fischer *et al.*, 1984] uses an ambiguous grammar for list productions to make it easy for the user to enter items at any position in a list. Since Poe is based on a structured editor, this ambiguous grammar is not used in a parser. Syned [Horgan and Moore, 1984] uses the text editing model for program parts, and completely reparses the syntactic construct that is being edited. Syned includes special processing for list productions to flatten the abstract tree representing recursively parsed lists, but this processing does not improve parse time.

2.2.3 Parallel Parsing

Early papers on parsing considered narrow classes of languages and developed mostly *ad hoc* techniques [Zosel, 1973; Huen *et al.*, 1977; Park and Burnett, 1979; Christopher *et al.*, 1981].

Of parallel parsing algorithms, those based on non-canonical generalizations of bottom-up strategies have been the most widely studied [Fischer, 1975; Mickunas and Schell, 1978; Cohen *et al.*, 1982; Ligett *et al.*, 1982; Cohen and Kolonder, 1985; Sarkar and Deo,

1986]. These algorithms construct their derivations starting at the leaves of the parse tree, building upwards. Because each level of the parse tree is constructed only after the deeper levels are completed, the depth of the derivation is the limiting factor in the execution time of these parsers. If deep parse trees can be avoided, these algorithms can make effective use of more processors.

Fischer's dissertation [1975] is a seminal work in parallel parsing that describes a synchronous parallel parsing algorithm for a vector machine. Cohen and Kolonder [1982] provide upper bounds for its speedup.

Schell [1979] considers the problem of scanning and parsing on an asynchronous multiprocessor, and develops a parallel generalization of standard bottom-up techniques. Ligett, McCluskey, and McKeeman [1982] describe a similar parallel parser based on a non-canonical parsing algorithm developed by Penello and DeRemer [1978] for error recovery. The performance of these and related algorithms have been analyzed using modeling [Cohen and Kolonder, 1985] and simulation [Schell, 1979; Ligett *et al.*, 1982; Sarkar and Deo, 1986].

2.3 Model and Assumptions

The inputs for syntactic analysis are the text file for a single compilation unit, a log of editing actions recorded by the editor, and the parse tree for the program prior to the editing changes. The inputs are assumed to be in memory when processing begins. Similarly, the parser output, a parse tree, is left in memory when processing completes.

It is assumed that a standard text editor has been modified to produce a log of user editing actions. The actions recorded in the log are simple insertions and deletions; other actions are translated into a sequence of insertions and deletions before inclusion in the action log.

Some simplifying assumptions are made about the scanner to clarify the exposition of the parsing algorithm. Specifically, we assume that editing actions do not affect token boundaries. This assumption excludes an editing action that deletes a separator between tokens, or inserts a separator within a token. A simple way of enforcing this restriction is to log editing actions on the basis of line numbers, and to disallow tokens that cross line boundaries (a simplification that has already been discussed as useful for lexical analysis). The textual positions of the beginning of all lines can be found in a prepass in $O(\log n)$ time using $O(n/\log n)$ processors for a file of length n using parallel prefix techniques [Ladner and Fischer, 1980], so it is easy to locate text in the source file given its line position.

2.4 Lexical Analysis

Lexical analysis is treated as a subroutine to syntactic analysis. This section considers the problem of implementing incremental lexical analysis on a shared-memory multiprocessor.

2.4.1 Scanning

A number of programming languages have a lexical structure that can be scanned much more efficiently than the general case. These are languages in which, by examining a constant number of non-initial contiguous characters of the input, it is possible to compute the state that a sequential FSM would end up in at the end of the substring. For example, if the language does not allow tokens to cross line boundaries (comments being terminated at the end of a line, as in Ada) and lines are bounded in length, then one may infer that the state of the FSM after the end of every line is the FSM's start state. In this way, each processor only need examine a constant number of characters of left context to be able to correctly scan its substring. This technique for inferring

FSM states is much simpler than implementing the full parallel prefix computation.

Instead of solving the general parallel-prefix problem in $O(\log n)$ time, this simplification makes it possible to solve the problem in $O(1)$ time using $O(n)$ processors, or with the same time and processor bounds as before but with an improved constant factor.

In the incremental case, substrings of new text can be scanned independently after determining the substring's initial state using the bounded left context, as above. In the case of more general regular languages, local changes can have far reaching effects, so incremental scanning can be as difficult as the non-incremental case.

On the basis of the simpler and more efficient implementation possible, we strongly recommend the use of languages whose lexical structure admit bounded regular left-context. This is one example of how compiler design techniques for parallel and incremental compilation impact language design.

2.4.2 Screening

In the overall compiler design, scanning is treated as a subroutine to parsing. When the tokens for new text are needed by a processor in the parsing phase, lexical analysis is performed on the new text at that time. This is inherently incremental because it only performs lexical analysis on new text. If we assume the languages lexical structure admits bounded left context, then lexical analysis for n characters using p processors can be performed optimally in time $O(n/p)$.

2.4.3 A Concurrent Extensible Hash Table

In a sequential compiler, the screener stores each identifier it finds in a search structure, so that all instances of the same identifier may be associated. In a parallel compiler, this

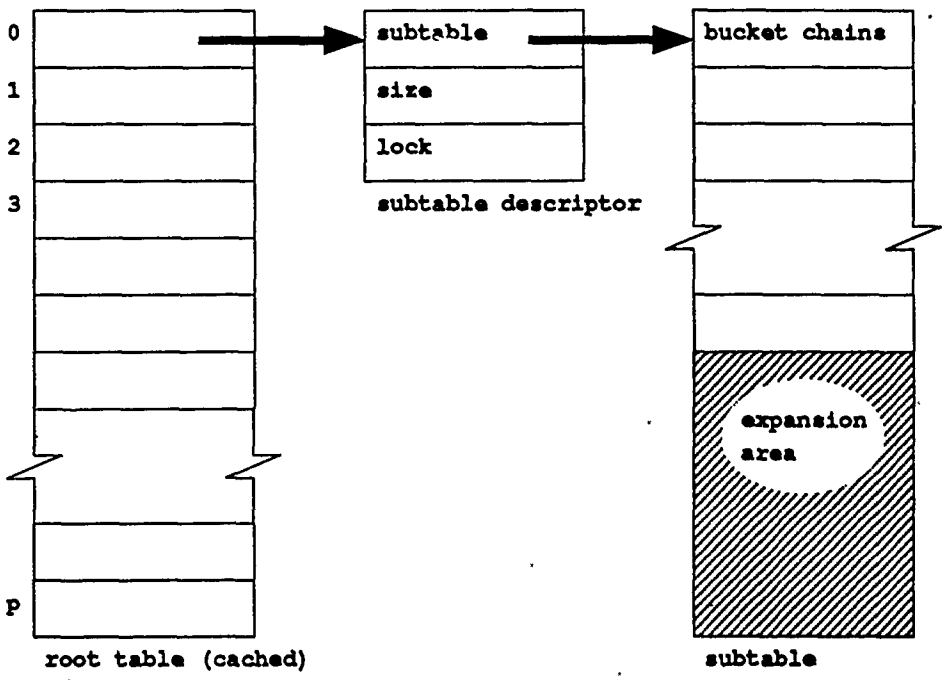

Figure 2.1: Two-level Concurrent Hash Table

search structure must allow a high degree of concurrency so that it does not constitute a sequential bottleneck.

Concurrent linear or extensible hashing schemes [Ellis, 1985] have been proposed as a suitable basis for a concurrent search structure, but they suffer from the disadvantage of having a single top-level lock to control expansion and contraction of the hash directory. This central could become a bottleneck or a point of memory contention. We describe an extension to these techniques that effectively distributes the centralized lock to reduce the source of contention while still performing in expected $O(1)$ time per operation.

Figure 2.1 presents a structure that can be used for just this purpose. A single global hash table of fixed size is used to index one of a number of subtables distributed among the processors. The number of buckets in the global table is proportional to the number of processors, so the probability of a conflict on the subtable lock will be some small constant. Each subtable contains an extensible hash table to contain keys that

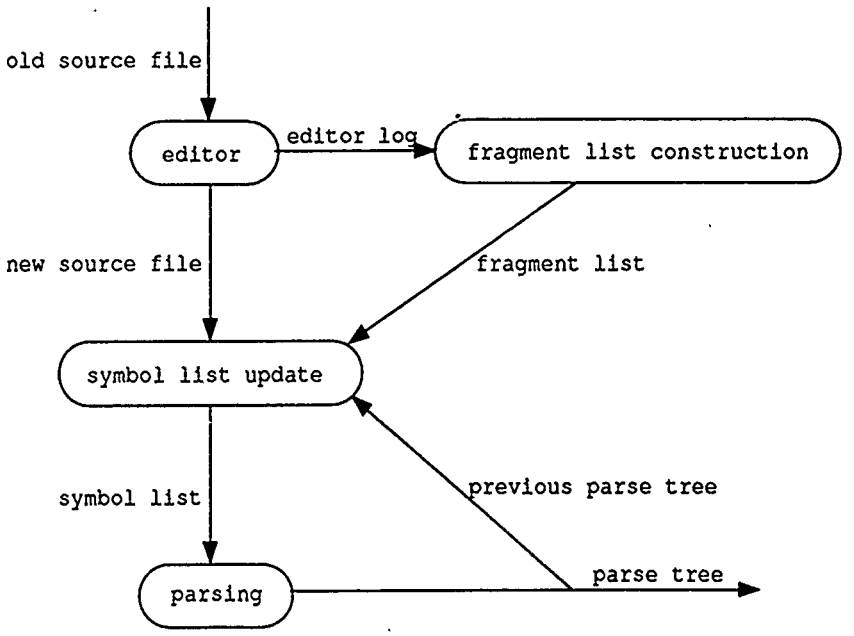

Figure 2.2: Overview of Parsing.

hash into its global bucket, so the subtable can be accessed in expected constant time independent of the size of the subtable.

Because the probability of a collision on the subtable descriptor is a small constant, operations on this hash table require expected $O(1)$ time. The space requirements for p processors and n operation is $O(p+n)$.

2.5 Syntactic Analysis

The task of the syntactic analysis phase is to translate a textual representation of the program into a parse tree. Because the compiler is incremental, it constructs the parse tree by modifying the parse tree for the previous version of the program. The log of editing actions produced by the editor is translated by the *fragment list construction* into a concise description of the difference between the old and new source files. The fragment list and the old parse tree are used during *symbol list update* to construct an updated symbol list for the program. The symbol list describes the source file as a

sequence of tokens from the source file and special *nonterminal symbols* that represent unchanged syntactic constructs; each contains the root of the old parse subtree for the construct. Finally, the new parse tree is constructed using a parallelization of conventional LR techniques. Figure 2.2 gives an overview of the syntactic analysis.

The strategy for symbol list updating will be to disassemble nodes of the old parse tree where the user has changed the text. A nonterminal symbol can be disassembled by replacing it with the nodes directly below it in its parse tree. Breaking up the parse tree in this way yields a list of symbols. From the list, symbols are deleted where the user deleted text, and symbols are inserted where the user inserted text, using lexical analysis as a subroutine. Finally, the symbol list is parsed to reduce sets of symbols into a nonterminal symbol, until only the start symbol of the grammar remains.

2.5.1 The Edit Log Data Structure

The edit log is an array of descriptions of editing actions in the order in which the user performed the operations. The log is output by the editor at the end of the editing session. Each action is an insertion or deletion, and for each change, the position and the size of the change is recorded. The positions are locations in the file in its transient state during editing, since this is the form in which the editor sees the file. The actual text manipulated by the editing actions does not appear in the edit log because the text file output by the editor contains the necessary information.

The edit log is a convenient data structure for the editor to output, because it is not much different from what the user types to the editor.

2.5.2 The Fragment List Data Structure

The text changes made during editing must be applied to the parse tree representation of the program to produce a symbol list that reflects the modified program. However,

each editing action refers not to positions in the original file, but to positions in the file in its transient state. To apply the changes to the tree in a highly parallel manner, the edit log must be translated into a fragment list, which is a concise description of the difference between the old and new source files only in terms of positions in the old and new source files.

The fragment list consists of descriptions of fragments of the source file in increasing order of (new source file) position. There are two types of fragments. An insertion fragment describes a contiguous segment of the source file that was inserted by the user, and records its length in the new text file. The position of the text in the new text file is implied by the position of the fragment in the fragment list. A "carry-over" fragment is a fragment of text that remains from the old source file, and is described by the position and length in the old text file. Deleted text is not described in the fragment list.

The concatenation of the fragments described in the fragment list is the entire file. The fragment list is thus a concise description of the new source file relative to the old source file.

There are two important invariants of the fragment list data structure: (1) there are never two adjacent insertion fragments in the fragment list, because such fragments are merged, and (2) the carry-over fragments appearing in the fragment list appear (to an in-order traversal) in increasing order of source position in the old source file. This latter condition results from the restricted form of editing actions that can be recorded, *i.e.* insert some lines and delete some lines.

Two of the data structures described here, the fragment list and the symbol list, are maintained using an applicative variant [Reps, 1982] of the concatenable queue data structure [Aho *et al.*, 1974]. Because the list is represented using a balanced tree, it can

be divided approximately in half in constant time. The concatenable queue supports the division of a list at a given position in $O(\log n)$ time, and allows two lists to be concatenated in $O(\log n)$ time where n is the number of queue entries.

The fragment list maintains summary information in the tree nodes of the concatenable queue. In particular, it must store the size of the text represented by the sublist, making it easy to locate fragments on the basis of source file information. The summary information can be maintained with only a constant factor overhead in parsing time.

2.5.3 Fragment List Construction Algorithm

The fragment list provides a differential description of the source file, much like the unix *diff* command output. The fragment list is computed from the edit log to avoid processing the entire source file.

The purpose of the fragment list is to concisely describe the source file, by indicating which parts are new and which parts are old. It is a list of descriptions of pieces or *fragments* of the file. Some translation is required to construct this list because each of the editing actions in the edit log refers only to transient positions in the source file.

The approach taken is parallel divide-and-conquer. The procedure *translate* translates an edit log into a fragment list by first translating two halves independently and then composing the results. As a practical matter, a more efficient sequential algorithm could be used once deep enough in the recursion to guarantee that all processors are kept busy.

Both the edit log and the fragment list represent a set of changes from a hypothetical beginning state. Thus, it makes sense to consider independently some portion of the edit log. The edit log is divided approximately in half to simulate the processing of two, shorter editing sessions. The second half of the edit log represents the set of changes relative to the state of the source file halfway through the editing session. The two

logs are then simultaneously translated into fragment lists. The first of these resulting fragment lists represents the hypothetical intermediate file relative to the original; the second fragment list describes the final file *relative to the hypothetical intermediate file.* Finally, these two file-to-file mappings are composed to produce the description of the final file relative to the original.

The procedure *translate*, below, translates the edit log into a fragment list using this divide-and-conquer technique.

```
procedure translate(edit-log el)
    returns fragment-list

(1) if el is only a single change, then
   (a) for a deletion at line k of m lines:
       return a fragment list with two fragments:
       (1) carry-over: position 1 and length k-1
           (if k=1 then discard this fragment)
       (2) carry-over: position k+m and
           length infinity (rest of file)
   (b) for an insertion at line k of m lines:
       return a list with three fragments:
       (1) carry-over: position 1 and length k-1
       (2) insertion with length m
       (3) carry-over at position k
           with length infinity.
```

When the input is sufficiently small, brute force is appropriate. Infinity is used to refer to the end of the file. Otherwise:

```
(2) divide the input edit log el into two lists
    el1 and el2 of approximately equal size.
```

Because the input is represented with an array an approximate split requires constant time. We translate the two resulting half-problems into fragment lists independently, and then compose the results.

```
(3) In parallel,
   (a) fl1 = translate(el1)
   (b) fl2 = translate(el2)
```

After this step, the fragment list *fl1* represents the state of the file halfway through the editing session. *fl2* represents the state of the output file *relative* to the state halfway through the session.

```
(4) fl = compose-fragments(fl1, fl2)
(5) return fl
```

The two fragment lists are composed by applying the set of changes described by one to the other. It is a simple matter to sequentially combine these fragment lists, splitting fragments in *fl1* at the borders of fragments in *fl2*. However, a sequential algorithm for n fragments would cause the running time of the translation to be $O(n)$, dominated by this merge step. It is therefore important to use a parallel composition algorithm.

The fragment list is independent of the order of the changes in its description of the file. As a result, the merge of *fl1* and *fl2* may be done in a highly parallel manner by dividing the problem at a source position boundary.

```
/* compose the text translation functions fl1 */
/* and fl2 by applying the changes described */
/* in fl2 to the file described by fl1 */
procedure compose-fragments(
      fragment-list fl1, fl2)
      returns fragment-list;

(1) if fl2 is a single fragment f
    then apply it directly:
   (a) if f is an insertion fragment, then
       b := fl2        /* to be returned */
```

When *fl2* is a simple insertion fragment as, it indicates that any text described by *fl1* has been deleted, since none has been carried over.

```
   (b) if f is a carry-over fragment, then
      (1) (b,a) =
          split-fragment-list(fl1, f.position)
      (2) (b,a) =
          split-fragment-list(a, f.length)
   (c) return b
```

A list containing only a carry-over fragment indicates that whatever was not carried over has been deleted. The appropriate segment of *fl1* is extracted by splitting it twice. The split-fragment-list function divides a fragment list into two fragment lists according to a source position, in which case only one part of the split is used. The summary information maintained at the nodes of the balanced representation of the fragment list record the length of the yield of each sublist, so split-fragment-list can be performed in $O(\log n)$ time where n is the number of fragments in the list to be split. If the split position resides inside a fragment, the fragment is divided into two fragments describing its halves.

When *fl2* is a single fragment brute force is used to compose *fl1* and *fl2*. Otherwise, *fl2* is divided into two lists.

```
/* otherwise, fl2 is not a single fragment */
(2) Divide the fragment list fl2 into two
    fragment lists a and b of approximately
    equal number of fragments.
```

The fragment lists are represented using a balanced tree scheme, so a balanced division will cost only constant time.

```
(3) In parallel,
   (a) c1 = compose-fragments(fl1, a);
   (b) c2 = compose-fragments(fl1, b);
```

These fragment list compositions can be executed in parallel because they occur in non-overlapping regions of the source file. For practical purposes, the fragment list fl1 could be split to reduce the running time, but this would have no effect on the asymptotic performance.

```
(4) return concatenate(c1, c2)
```

A description of the entire source file is constructed from the description of its two halves. If the last node of *c1* and the first node of *c2* are insertion fragments, they will need to be combined. Concatenation is one of the operations that the balanced data structure supports in $O(\log(n+m))$ time for lists of length n and m.

2.5.4 The Symbol List Data Structure

The symbol list data structure represents a list of tokens. There are two types of symbols: a terminal symbol represents a token appearing directly in the source program, and a nonterminal symbol represents a syntactic construct that has already been recognized. Each nonterminal stores the root of a parse tree for its syntactic construct, and replaces its yield in the symbol list. An invariant of the symbol list data structure is that the concatenation of the yields of the symbols in the list is exactly the source program. In this sense, the symbol list is similar to a sentential form.

The symbol list is maintained using an *applicative* variant of the concatenable queue data structure, similar to that used for the fragment list. The concatenable queue supports the division of a list of n elements at a given position in $O(\log n)$ time, and allows two lists of length n and m to be concatenated in $O(\log(n+m))$ time. The elements of such a list can be enumerated in linear time.

2.5.5 Symbol List Updating Algorithm

The next task of the parser is to produce a symbol list for the program. Since the compiler is incremental, the previous symbol list is updated using the fragment list. The old symbol list contains only the start symbol of the grammar if the previous parse was successful; in this case it is called a complete parse tree. There will be more than one symbol in the old symbol list if there was a syntax error in the previous version of the source file.

Each nonterminal in the symbol list contains the parse tree for the construct, so the list may be expanded by replacing any nonterminal in the list by the list of symbols directly below it in the parse tree. The yield of a symbol list is not changed by such expansion.

To update the symbol list to represent the new program, any nonterminals that span source positions where changes have been made by the user are repeatedly *disassembled*. A nonterminal symbol is disassembled by replacing it in the fragment list with the symbols below it in its parse tree. Next, some symbols are removed from the list and some are inserted to update it to represent the new version of the source file. The hope is that the unchanged text will have the same syntactic derivation as before; in this case the parser can avoid repeating previous parsing work if the previous work is valid in the new parse tree. When the derivation is different, the nonterminal must be disassembled.

A parallel algorithm to update the symbol list is also based on divide-and-conquer. We split both the fragment list and old symbol list, and recursively update the two halves. The two updated halves are then concatenated to produce the symbol list for the new text file.

```
procedure update-symbol-list(
   symbol-list t;
   fragment-list fl;
   /* position of result in new file */
   integer position;
   /* position of symbol list in old file */
   integer offset)
   return symbol-list;

(1) if the fragment list contains only the
    single fragment f
   (a) if the fragment is an insertion fragment,
       then run a scanner on a portion of the
       program file at "position" for length
       f.length+offset.  return the resulting
       list of token symbols.
```

If the fragment list contains only a single insertion fragment, then all of the text is new. The source file is scanned at the appropriate position. We assume a parallel scanning algorithm that runs in $O(\log n)$ time for text of length n.

```
(b) if the fragment is carry-over, then
    (1) (a,b) :=
        split-symbols(t, f.position-offset)
    (2) (b,c) := split-symbols(b, f.length)
    (3) return b;
```

If the fragment list contains only a single carry-over fragment, then the symbol list can be extracted from the old symbol list. The old symbol list is split twice to extract the appropriate section.

The split-symbols procedure is a hybrid of the split operation on the symbol list and the disassembly of the nonterminal nodes discussed above. It splits a given list at a given source position relative to the beginning of that list. If the source position lies within a nonterminal, the symbol is repeatedly disassembled until the position is between symbols in the list. The list is then split between these symbols. Note that source length information is maintained in each symbol and summarized in the nodes of the balanced tree representation of the list, allowing location of the position where the list is being split in $O(\log n)$ time where n is the size of the token list. Time proportional to the depth of the parse tree is also required to disassemble the nonterminals. The result of split-symbols is two symbol lists.

```
(2) Split the fragment list fl into two halves
    fl1 and fl2 of approximately the same size.
```

The problem is divided into two smaller subproblems by dividing the fragment list which describes the source file. Since the fragment lists are represented using a balanced tree scheme, only constant time is required.

```
(3) Find the first carry-over fragment f in fl2
    Let p = f.position
```

This step finds the position in the source file where the old symbol list may be split. Each half of the recursion will be used for one of the subproblems.

Because two insertion fragments are never adjacent, p will be the first or second fragment. If *fl2* is only a single insertion fragment, use the position *t.length* as p.

```
(4) (t1, t2) := split-symbols(t, p)
```

The old symbol list is split at the position boundary found in (3). Because carry-over fragments appear in increasing order of position in the original source file, pieces of the source file that the previous and current versions have in common appear in the same order. This fact allows the symbol list to be split along with the fragment list, so the recursive calls can use smaller symbol lists as input.

```
(5) In parallel,
   (a) t1 := update-symbol-list(
         t1, fl1, position, offset)
   (b) t2 := update-symbol-list(
         t2, fl2, position+fl1.length,
         offset+t2.length)
```

The two parts of the resulting symbol list are constructed independently from the fragment lists describing them. Changes are recursively applied to right and left halves of the old symbol list.

```
(6) return concatenate-symbols(t1, t2)
```

The two parts of the updated symbol lists are concatenated to produce the symbol list for the entire new file.

2.5.6 Parsing Algorithm

Having constructed the symbol list for the new program, the final problem is to turn it into a parse tree for the entire program. As with all bottom-up parsing, a list of symbols representing a single syntactic construct is replaced by a single nonterminal. This reduction replaces the group of symbols with the corresponding enclosing syntactic construct, constructs a tree node for it, and inserts the representative nonterminal into the symbol list. Reductions must be applied repeatedly until only the symbol for the start symbol remains.

Conventional LR techniques perform only the leftmost reduction at every step, reducing only constructs that appear on the top of the parsing stack. These techniques are inherently sequential. Schell [1979] describes a non-canonical generalization of LR parsing in which a number of processors simultaneously work on different parts of the input. In Schell's algorithm, the input string is divided into a number of sections, and each section is parsed by a separate processor. Each processor independently finds locally leftmost phrases to reduce as in conventional LR parsers. When it is unable to continue because of an error, inadequate state, or insufficient stack depth for a reduction, a processor passes the terminal and nonterminal symbols in its stack to the processor working on the segment to its left to be treated as a continuation of its input. Thus, the parser is a pipeline of processors. The LR parsing techniques are easily generalized to allow the resulting nonterminal symbols.

The parser completes when the leftmost processor completes parsing the entire program. Because much of the program is "pre-reduced" by other processors, the leftmost processor potentially sees many fewer symbols than a sequential parser would.

In the parallel parser, finding a phrase is difficult because the non-leftmost processor may not have sufficient context to determine the leftmost phrase. A further complication

arises when a processor does not have the prefix of a phrase on the parse stack when a reduction is attempted. These are the two fundamental problems solved by Schell by modifying the state construction process and the parsing actions.

Since only the leftmost processor knows its context in the parse of the entire string, the non-leftmost processors start in a *super-initial* state, which is constructed by including all non-initial items. This state reflects the fact that these processors start without the benefit of left context normally provided by the contents of the parse stack—it may be starting out at an arbitrary position in any syntactic construct. The closure and parser actions are constructed as usual.

During the construction of the states, some of these additional states may display shift-reduce or reduce-reduce conflicts on certain inputs. A processor cannot distinguish the phrase to be reduced, and will pass the nonterminals on the parse stack to its neighbor to the left, which presumably has more context. In this case, Ligett *et. al.* [1982] restart the parser in the super-initial state on its remaining input. The more refined continuation states constructed by Schell [1979] reflect knowledge gathered on the stack prior to the conflict, and is therefore used for the construction of our parse tables.

The problem of insufficient stack depth to perform a reduction is handled by appending the symbols on the parse stack of the current processor to the input for the processor on the left, and restarting the current processor in the state accessed from the super-initial state by shifting the left hand side of the production. This simplification of the algorithm described by Schell has little effect on the running time.

The parsing algorithm of Schell will not make a reduction that is an incorrect derivation for the input string, so each nonterminal seen is part of the correct derivation. However, our incremental parser may start with nonterminals from the old parse tree

in its input. These nonterminals do not necessarily appear in the correct derivation of the new program, and might therefore have to be disassembled during the parse. Even worse, these incorrect nonterminals may be shifted by a processor, allowing the parser to make incorrect assumptions about the syntactic structure of the input. These problems and their solution are the main differences between the parsing algorithm of Schell and that described here.

For example, consider the following grammar:

```
S   ::= "PASCAL" SP | "ALGOL" SA
SP  ::= ...(a pascal grammar)...
SA  ::= ...(an algol grammar)...
```

In this example, the interpretation of the entire string rests on the first token. But if only the first token is changed, the nonterminals for the rest of the program will not be broken up by the symbol list update phase of the parser. This language is still SLR(1), for instance, if *SP* and *SA* are SLR(1).

A processor working on a segment in the middle of this program will enter a subset of the parse states that include only Pascal items as soon as it shifts a Pascal-only symbol. Unfortunately, the program may well be an Algol program. While this does not affect the correctness of our algorithms, it does affect their efficiency. This language displays a kind of non-loca that makes parallel parsing difficult; languages that display locality are easier to pars , and are easier for people to read because there are frequent hints to the reader about the context.

Under the above scenario, an incorrect nonterminal symbol will appear eventually to a processor as an error entry in the parse tables. It is not a true error because it does not necessarily represent an incorrect program.

In order to efficiently distinguish between the various causes of finding and error entry in the parse table, each parse tree node records the leftmost token in its yield.

The grammer is constrained to allow no ϵ-deriving nonterminals. Now, when an error occurs, the processor can distinguish these three cases:

- if the input symbol is a nonterminal, and there is a valid action in the current state for the first token of its yield, then the reduction represented by the input nonterminal is not one the current parser would have chosen. The reduction was made in the old parse tree or by a parser to the right. The input symbol is therefore disassembled and prepended to the current processor's input.

- if the input token is a terminal t, or a nonterminal with t as its leftmost terminal symbol, and the top of the stack is a nonterminal n, and and $t \notin FOLLOW(n)$ then the top of the stack is not a reduction that would have been chosen by the local SLR(1) parser. Therefore the top of the stack is popped, disassembled, and prepended to the input.

- in any other case, an invalid nonterminal symbol has been shifted earlier (or possibly the program is in error). The symbols on the stack are passed to the processor to the left, and the current processor resumes parsing starting with super-initial state on the remaining input. (The idea is to pass the problem on to a processor which is likely to have more context with which to solve it.) If this is the leftmost processor, then a genuine error has been recognized, and normal error recovery is invoked.

The correctness of the parsing algorithm follows from the correctness of Schell's [1979] parallel parsing algorithm. The only difference is the existence of incorrect nonterminals in the input symbol stream. The proof assumes the grammar is SLR(1) and the input program is in the language.

Theorem 1 *The parsing algorithm presented here constructs a correct derivation for the input program.*

Proof: Because the leftmost processor breaks up old nonterminals from the top of the stack when it detects an error, the parse will be correct if the leftmost processor never lets an erroneous nonterminal get past the top of its parse stack. By contradiction, it can be shown this will never happen. Assume the leftmost processor shifts some symbol s onto the stack above the erroneous symbol t while parsing a correct program. Let w be the symbols corresponding to the stack contents below t. Then wts is the prefix of a sentential form (the prefix of some valid program in the language). The language is unambiguous because the grammar is SLR(1), and there are no ϵ-deriving nonterminals, so a sequential parser would construct the same derivation. This contradicts the assumption that the symbol t is incorrect. □

2.5.7 List Productions

One rough estimate for a lower bound on the parsing time is the depth of the newly constructed portions of the parse tree. While the nesting structure of typical programs is probably small, the parse tree can be much deeper due to syntactic lists. The straightforward method for parsing lists induces a linear depth parse tree, and this will result in a linear parsing time for a program consisting entirely of a single list.

The standard way to represent lists of items in an LR-based grammar is using a left-recursive rule:

```
<list>   ::= <item>
<list>   ::= <list> <item>
```

This causes the parser to construct a syntax tree that is linear depth in the length of the list. This is unfortunate for our parallel parser, because its execution time has

a component that is linear in the depth of the newly constructed portions of the parse tree. We expect that syntactic lists are a common cause of deep parse trees in typical computer programs. To improve parsing efficiency for long lists of syntactic items, we allow special *list productions.* All right and left-recursive lists may be replaced in the grammar by an application of this special rule form and a few "helper" productions. For example, the productions

```
<stmt>      ::= BEGIN END
<stmt>      ::= BEGIN <stmt_list> END
<stmt_list>     ::= <stmt>
<stmt_list>     ::= <stmt_list> ';' <stmt>
```

may be replaced by

```
<stmt>          ::= BEGIN END
<stmt>          ::= BEGIN <stmt> END
<stmt>          ::= BEGIN <stmt_list> <stmt> END
<stmt_list>     ::= <stmt> ';'
<stmt_list>     ::= <stmt_list> <stmt_list>
```

In general, list productions are specified as follows:

```
<list>      ::= <list> <list>
```

This form is easily recognized by the parser builder as a special case. It describes a syntactic list that derives one or more of the syntactic item on the right hand side. We do not allow zero items because our parsing algorithm assumes that the grammer is constrained to allow no ϵ-deriving nonterminals. While more complicated forms of list productions might be allowed, this restricted form allows a fairly simple correctness proof and analysis.

Although this production is ambiguous, it has the desirable property that is describes all binary syntax trees with the items at the leaves. The strategy for efficient parsing will be to maintain these trees internally as *balanced binary trees* [Myers, 1984], as if

the parse had constructed a fully balanced derivation. To a large extent, the increase in efficiency will be due to the ability of the parser to actually construct a balanced derivation.

Of course, we must deal with the ambiguities that are introduced by the productions. These new productions are handled during state construction exactly as other productions, with the following exceptions:

- The item

  [<list> ::= <list> • <list>]

  is *not* included in the super-initial state. This allows the local processor to fully process all adjacent items appearing in its input.

- If a state contains the item

  [<list> ::= <list> <list> •]

  then the item

  [<list> ::= • <item>]

  is removed from the state. This resolves a potential shift-reduce conflict and allows the local processor to fully reduce its items as would a sequential left-recursive parser.

- During parsing, a reduction by the rule

  [<list> ::= <list> <list>]

  causes the *join*, or concatenation, of two balanced structures, and results in the creation of a single balanced tree structure. The resulting abstract parse tree does not necessarily represent the actual derivation which the parser evaluated due to the rebalancing that may occur, but the resulting parse tree remains a valid representation of the list.

These rules allow a local processor to fully reduce all adjacent items appearing in its input into a single list nonterminal, and resolve all ambiguities introduced by the list productions. By an extension of the proof of correctness in previous sections, it.can be shown that the resulting parser correctly parses only strings in the language.

2.6 Error Detection and Recovery

The error recovery algorithm described by Graham and Rhodes [1975] and later refined by Penello and DeRemer [1978] is appropriate for error recovery in our parser, since the required "recovery states" are a subset of the states that were added for parallel parsing by Schell [1979].

Because the parser is incremental, it is important to maintain the correspondence between the parse tree and the program text. Therefore, if tokens are inserted into or deleted from the symbol list as a result of error recovery, the source file should similarly be modified.

Alternatively, a change can be made only temporarily on a copy of the symbol list, in which case a syntax error detected in the parsing phase will result in a symbol list that has not been fully condensed to the start symbol. Since the symbol list is represented using an applicative data structure, the copy can be created and modified with little overhead. This unmodified symbol list can still be used without any problem as the old parse tree for the next compilation.

2.7 Complexity Analysis

We consider the processor and time complexity of the algorithms described in this chapter, as well as the processor-time product, since that is a measure of the parallel efficiency, or speedup, that may be observed in the asymptotic case. We demonstrate

that our algorithms have polylogarithmic execution time and an optimal processor-time product.

2.7.1 Fragment List Construction

Theorem 2 *The time required for the procedure* `translate` *is* $O(\log^3 m)$ *in the length of its input, the edit log.*

Proof: The *split-fragment-list* function requires $O(\log f)$ time where f is the number of fragments. The procedure *compose-fragments* recurses to a depth of $O(\log f)$, performing a constant number of split or concatenation operations at each step, giving a running time of $O(\log^2 f)$. The full translation recurses to $O(\log m)$ depth, where there are m editing actions.

Because each editing action generates at most two fragments, $f = O(m)$. Translation uses a constant number of applications at each recursion level, so the parallel running time of the entire translation algorithm is $O(\log^3 m)$. □

Notice that the complexity is independent of the size of the associated program source file.

Theorem 3 *The maximum number of processors used in* `translate` *is* $O(m)$ *in the size of the edit log.*

Proof: The proof is by induction on the size of the fragment list. For $m = 1$, the procedure computes the result directly and returns, using only 1 processor. For $m > 1$ edit actions, the edit log is divided into two smaller lists of size j and k, where $j + k = m$. The recursion in step 3 uses $j + k$ processors by the induction hypothesis, and *compose-fragments* uses $O(k)$ processors by a similar argument, so the total number of processors used is $O(m)$. □

Theorem 4 *The procedure* `translate` *can be performed in* $PT = O(m \log m)$, *which is optimal.*

Proof: It can be shown by problem reduction from the (integer) sorting problem that a sequential implementation of fragment list construction requires $O(m \log m)$ time. Consider a sorting problem in which the input is a list of integers $(a_i), 1 \leq i \leq m$ and in which no two a_i are equal. From this an edit log of length $2n$ can be constructed in $O(n)$ time in which a_i is translated into two edit actions - delete line at position a_i followed by insert line at position a_i. After this is translated into a fragment list, the fragment list can be translated in linear time into a solution to the sorting problem. Each carry-over fragment is ignored, and each insertion fragment describes a contiguous range of integers which should be output. General sorting is known to require $O(m \log m)$ sequential time, so fragment list construction also requires $O(m \log m)$ sequential time.

We can reduce the processor requirement of the parallel algorithm by processing $O(\log^2 m)$ editing actions sequentially on each of $O(m/\log^2 m)$ processors, and use the parallel algorithm only to combine these results. Using an obvious sequential algorithm locally, each processor can translate its editing actions in $O(\log^2 m \log\log m)$ time, and the time required to combine the results remains $O(\log^3 m)$. Thus, when $O(n/\log^2 m)$ processors are used the processor-time product is $O(m \log m)$, which is the same as the sequential time and is therefore optimal. □

2.7.2 Symbol List Update Algorithm

Definition 1 (Tree Depth) *Let* $d(n)$ *be the depth of the parse tree.*

There are two main factors that contribute to the depth of the parse tree: the nesting depth of the program and the length of lists appearing in the program. A single list of length n will generate a balanced tree of depth $O(\log n)$. A program with nesting

depth k cannot be parsed by techniques that construct the parse tree botton-up in less than $O(k)$ time. We will assume that the tree depth is some function $d(n)$ of the length of the program, and parameterize the analysis in terms of this function. Ideally, $d(n) = O(\log n)$.

The complexity of updating the symbol list depends critically on the depth of the parse tree. Because list productions produce more balanced and therefore shallower parse trees than conventional grammars for lists, their use can improve the performance of the symbol list construction.

Theorem 5 *The time complexity of symbol list construction is* $O(\log m(\log n + \log m + d(n)))$.

Proof: The *update-symbol-list* algorithm recurses to a depth of $O(\log m)$ in step 5 when there are m fragments in the fragment list. Step 1 takes $O(\log n)$ time in the length of the program and step 2 takes constant time. Step 3 takes $O(\log m)$ time and step 4 takes $O(d(n))$ time. The time complexity of symbol list construction is therefore $O(\log m(\log n + \log m + d(n)))$, which dominates the total running time of the symbol list update. □

If we assume $d(n) = O(\log n)$ and $m = O(n)$, this gives a running time of $O(\log^2 n)$.

Theorem 6 *The number of processors used in* update-symbol-list *is* $O(n)$.

Proof: The proof is by induction on the length of the fragment list. For $n = 1$, the procedure uses only one processor. For $n > 1$, step 2 divides the fragment list into two parts of length j and k, where $j + k = n$. The recursion in step 5 uses $j + k$ processors by the induction hypothesis, so the total number of processors used is $O(n)$. □

2.7.3 Complexity of Parsing

Schell's parsing algorithm requires $\Omega(\sqrt{n})$ time because no processors drop out of the computation. We require that the parse takes place in $O(log_2 p)$ distinct phases when using p processors. Between phases, every other processor will drop out of the computation and pass the symbols from its stack to the remaining active processor to its left. In the final phase, the remaining processor, which is the processor that began on the initial symbols of the program, will complete the parse with the final symbols of the program.

Upper bounds for speedup in parallel parsing based on modeling [Cohen and Kolonder, 1985] and simulation [Sarkar and Deo, 1986; Ligett *et al.*, 1982; Schell, 1979] would appear to be directly applicable to our algorithm. Two crucial constraints are the depth of the parse tree and the *locality* of a language, discussed in Section 2.5.6, which may have a large impact on the actual efficiency. We will parameterize the parsing complexity in terms of these.

In addition, the extent to which constructs from the previous parse tree can be used without modification affects the parsing speed in the parallel *incremental* case. The rest of the analysis here is for the parallel, non-incremental case. We begin with a definition of the other major factor limiting parallelism of this parsing technique.

Definition 2 (Locality) *Let l be the* locality *of the program being compiled: this is the number of correct reductions that a processor, given an arbitrary segment of the program as input, will fail to make (because of insufficient context) but for which it has all of the symbols in its input.*

The locality is as much a property of the language as of the program. A language discussed in the next chapter has $l = O(1)$ locality for every string in the program. We will treat locality as some function $l(n)$ of the length of the program.

Each processor is able to fully reduce its subset of symbols in a list once it is past the initial $O(l(n))$ symbols, excluding the final symbol, for which the local processor may not have the look-ahead to resolve a conflict. The processor requires time proportional to the length of its input multiplied by $O(\log n)$ to balance a possible list upon each reduction by a list production. We need to bound the length of the fully reduced portion of the symbol list for the complexity analysis.

Lemma 1 *The minimal exact cover of a contiguous set of k leaves in a tree of size n and bounded fanout is of size $O(d(n))$, where $d(n)$ is the depth of the tree.*

Proof: Consider the set of nodes on a path through the tree from the first to the last leaves in the set. All nodes on the minimal cover come either from this set or direct children of this set. The path length is $O(d(n))$ and the number of children is $O(d(n))$ because the tree is bounded fanout. □

Lemma 2 *At the end of each phase of parsing, each processor will have $O(s(n))$ symbols remaining on its stack, where $s(n) = d(n) + l(n)$, a measure of the two main sequential constraints to this method of parsing.*

Proof: By Lemma 1, a processor that can make all reductions on its input symbol list will leave a list of size $O(d(n))$. Each correct reduction decreases the number of symbols in the list by a constant number equal to the length of the right hand side of the production by which the parser is performing the reduction. Thus, failing to reduce by $l(n)$ productions generates $O(l(n))$ extra symbols in the reduced symbol list. □

The size of the symbol list at the end of each phase is also the size of the input symbol list that each processor must process at the beginning of each non-initial parsing phase. Given $p(n)$ processors, we can compute the time required for each processor to process

its portion of the input sequentially, and the time required for each of the $O(\log p(n))$ merging stages.

Theorem 7 *The time to parse a program of length n with sequential constraint $s(n)$ and using $p(n)$ processors is $O(n/p(n) + s(n)\log n \log p(n))$.*

Proof: The first stage requires each processor to process $O(n/p(n))$ tokens of the input program. Each reduction requires $O(\log n)$ time because it may be a list production and require balancing, so the total time is $O(n \log n/p(n))$. However, we can reduce this time by failing to balance at every reduction, and only balancing the final version of each list. A balanced binary tree can be constructed sequentially in time $O(k)$ from a list of length k, so this reduces the time complexity of the first phase to $O(n/p(n))$.

The remaining phases are more interesting. When two processors combine their symbol lists, the processor on the left continues its parse on the symbols remaining from the stack of the processor on the right. The time to do this is bounded by the time to balance a possible list production for each symbol in the list, that is $O(\log n)$, multiplied by the number of symbols in a list, which is $O(s(n))$ by Lemma 2. Thus the total time for one merge stage is $O(s(n)\log n)$. There are $O(\log p(n))$ merge stages, so the total parsing time after the first stage is $O(s(n)\log p(n)\log n)$. □

Corollary 1 *If $s(n) = O(\log n)$, and $p(n) = O(n/\log^3 n)$, we obtain $T(n) = O(\log^3 n)$, and the processor-time product $PT(n) = O(n)$, which is optimal.*

This shows that if we can bound the locality and tree depth, we can choose a bound on the number of processors that allows us to parse with an optimal processor-time product.

3 Parallel Separate Compilation

At first glance, parallel separate compilation would seem to be unrelated to parallel incremental compilation. In fact, parallel separate compilation is an important technique that effectively utilizes parallelism in incrementally compiling a program composed of more than one source unit.

In addition, techniques for implementing the specialized semantics of the *makefile* language are presented as simplified examples of algorithms that will be further developed in the following chapter on general programming language semantics. Specifically, this chapter illustrates the use of list productions in language description, discusses the representation of a symbol table, presents algorithms for efficiently determining what has changed after a reparse in order to update the symbol table, and discusses error reporting and recovery.

3.1 Introduction

Many programming languages have constructs that allow a single program to be expressed as a number of separate source units. Some languages, such as FORTRAN and its variants, do not perform type checking across source unit boundaries. For these languages a great deal of large-grain parallelism in compilation may be exploited by compiling all of the source files simultaneously. The amount of parallelism that we may

exploit is influenced solely by the organizational decisions made by the program author, but fortunately typical large programs are composed of a many of source files.

We can avoid compiling each source file that has not changed since its last compilation to achieve incremental processing at a very course grain. For FORTRAN, we may simply compare the date that the source file was last modified with the date that the object file was written to decide if a compilation unit needs to be recompiled. This is incremental compilation at a very course grain.

The more interesting cases are languages in which source files or compiled sources (object files) may be interdependent. The programming language `C`, for instance, supports *include* files that are textually included into the source program. Since include files may include other files, this leads to an arbitrarily deep dependency graph. However, it is only the `C` source files that need to be compiled, and in general they may all be compiled simultaneously because they are not interdependent.

A more interesting example is provided by the **Mesa** compiler and by some implementations of **Ada** and **Modula-2**. For these systems, special *definitions* or interface source modules specify a strongly-typed interface between an implementation module and its clients. The interface is compiled separately from the implementation. The compiler reads the compiled interface files when compiling a package that imports its definitions. Both the implementation of the package and any modules that depend on it must be compiled after the interface because the compiler uses the type information extracted from symbol tables in the precompiled interface.

The compilation of multi-source programs in these languages, and many other computational tasks that are composed of a number of processing steps with intermediate results, may be described in the specialized *make* [Feldman, 1979] language. A *makefile* (a *make* source file) describes an acyclic dependency graph among a set of files, and

describes how to construct each dependent file from its prerequisites. This dependency graph is an instance of a dataflow graph, a directed acyclic graph with values computed at each node and edges representing dependencies. This chapter considers how a dataflow dependency graph may be constructed incrementally and in parallel from its description in a *makefile* and how the graph may be evaluated (*i.e.* source files compiled) incrementally and in parallel.

In the next chapter a dataflow graph is created by another technique: an attribute grammar for a particular parse tree. The symbol table data structure described here is used as a basis for symbol tables there.

3.2 Related Work

The make utility [Feldman, 1979] is a method for maintaining programs composed of multiple source files. **Make** takes as input a *makefile* describing the dependency graph among components of the system, and rules or commands for reconstructing components once the prerequisite files have been constructed. When it is used to recompile programs after changes, the **make** utility represents incremental compilation at the granularity of a source file. *Makefiles*, or *make* source files, are written in a special language designed to concisely express the dataflow graphs that arise in separate compilation.

The problem of optimal parallel scheduling of dataflow graphs is NP-complete [Ullman, 1976; Garey and Johnson, 1979]. However, it is known that an arbitrary schedule will evaluate the graph within twice the time of an optimal schedule [Graham, 1976]. This optimization problem continues to be pursued by the operations research community, but because it can affect the time complexity by at most a constant factor, we do not consider the problem in detail.

The idea of a parallel *make* facility is not new. A number of multiprocessor manufac-

```
program:        main.o subroutines.o
        ld -o program \
                main.o \
                subroutines.o -lc

main.o:         main.c subroutines.h
        cc -c main.c

subroutines.o:  subroutines.c subroutines.h
        cc -c subroutines.c
```

Figure 3.1: Simplified Example of a Makefile

turers supply concurrent *make* utilities as a standard part of their software distribution. In addition, the Free Software Foundation's *make* utility has hooks for concurrency. Unfortunately, little has appeared in the research literature describing these programs or their underlying algorithms.

3.3 The Makefile language

The *make* utility can be thought of as a compiler for a specialized language, used to describe the dataflow graph inherent in compiling a multi-source program. We consider a simplified version of the *makefile* language and show how it can be processed in parallel and incrementally. This is a matter of maintaining and scheduling the dataflow graph.

A *makefile* program (hereafter called a makefile) is composed of a number of rules. Each rule describes a single target file: the file is named, followed by a colon and a list of files on which it depends. Subsequent lines give commands used to construct the target from its prerequisites; this list of commands ends at an empty line. Figure 3.1 is a simplified example makefile that describes how to construct a C target program from its two sources and a single include file.

Lexically, the makefile language is very simple. Tokens are contiguous sets of print-

able characters separated by spaces or tabs. A newline serves simply as a token separator, like a space, when escaped with a backslash; otherwise the newline is a special token. Each rule ends with a blank line. We arbitrarily limit lines to 80 characters in length. Because comments do not cross line boundaries, and because source lines have a constant length bound, we can use the simplified scanning algorithm of the previous chapter that starts scanning at line boundaries to scan in constant time using a linear number of processors.

3.4 Parsing the Makefile language

The syntactic structure of the language is straightforward. Figure 3.2 gives a grammar for the Makefile language that exploits list productions; the terminal symbol NL represents a newline. With the exception of the special productions to represent lists, this grammar is nonrecursive. Therefore, a makefile can have at most a constant nesting depth, clearly within the $O(\log n)$ nesting depth assumed for the parsing time analysis.

Furthermore, this grammar illustrates a strong form of grammatical locality that is also useful: every adjacent pair of IDs appearing in the input can be reduced to an `<ID List>`. The only inadequate states in which a parsing processor will not be able to proceed will be due to conflicts between the right hand side of the `<Command>` production and the middle of the `<Rule>` productions (after the symbols `<ID List> NL`). In this case a processor will pass at most two symbols to the processor to its left. Because the processor will continue in a state that is also in the sequential SLR(1) automaton, the node will never again encounter an inadequate state. Similarly, because of the nonrecursive grammar, a reduction with insufficient stack can only happen a constant number of times, each time passing a constant number of symbols to the parser on the left. Therefore this makefile language can be parsed in parallel as efficiently as a simple list, in $O(\log^2 n)$ time using $O(n/\log^2 n)$ processors in the worst case.

```
<Start>
        ::=     <EOF>
        |       <Rule>
        |       <Rule List>
<Rule List>
        ::=     <Rule List> <Rule List>
        |       <Rule>
<Rule>
        ::=     ID ':' <ID List> NL <Command List> NL
<Command List>
        ::=     <Command List> <Command List>
        |       <Command>
<Command>
        ::=     <ID List> NL
<ID List>
        ::=     <ID List> <ID List>
        |       ID
```

Figure 3.2: Grammar for the Makefile language

3.5 Parallel Makefile semantic analysis

As well as being syntactically simple, the Makefile language is semantically simple. Each rule defines a semantic attribute that is a software component or file to be constructed, names those other components upon which it depends, and provides code to bring it up to date when the prerequisites change.

For each component of the system the *makefile* contains a rule that appears as follows:

```
component: prereq1 prereq2
        commands
```

This rule says that in order to construct `component`, first the prerequisite files `prereq1` and `prereq2` must be constructed. Then, the given commands will construct the file `component`. It is the responsibility of the author of a makefile to ensure that the

commands do in fact create the file named component or update the file's recorded last time written. This requirement is needed because the **make** utility uses the file's last time written to determine what software components need to be reconstructed after a change. If a component is newer than its prerequisites, and its prerequisites are up to date, then the component is considered up-to-date.

The *makefile* describes a dependency (directed) graph. To be correct, the graph must be acyclic. The usual implementation of a **make** utility performs a depth-first traversal of the dependency graph described by the makefile, starting at a user-specified *root* component, to make the component up-to-date. At each component, the utility (recursively) ensures that the prerequisites are either leaves or are up-to-date. The leaves of the dependency graph are the true source files in the system, for which there are no prerequisites. If any of the prerequisites are more recent than the component being considered, the component is reconstructed using the commands given in its rule.

There is some freedom in this description because there is more than one depth-first ordering. The most common ordering used in implementations of **make** is a sequential enumeration of prerequisites in the order in which they appear in the rule. However, any topological ordering of components could be used to reconstruct a software component.

A parallel implementation of make may take advantage of this freedom from sequential constraints. The strategy is to schedule the processing of components using an algorithm analogous to topological sort. As an optimization, the scheduler can consider only components and dependencies that are visited by a transitive closure of changed source components. This optimization is more important later, for semantic processing of general programming languages, because the reduction in work is more significant due to the finer grain of the attributes.

The dependency graph may be constructed in two passes. The first pass inserts all

of the (*name, component*) pairs into a single symbol table. Each component's semantic value may be represented in the table as the current time stamp of its file. The second pass will read the rule lines and link up references with their corresponding definition, thereby constructing the dependency graph within the symbol table. The second pass does not need to wait for symbol table entries to become defined, nor insert them, because the symbol table (except for the dependent lists) has been constructed fully in the first pass. The second pass creates the dependent list for each component, thereby creating a representation of the dependence graph. The dependence lists are represented as a parallel search structure.

When this second pass finishes, the dependence graph is complete and may be used to schedule component evaluation. Parallel transitive closure is used to identify all components affected by changed components or rules in time dependent only on the depth of the affected dependency subgraph. (Of course, we would need some way to efficiently identify which source components have changed; this is discussed later.) Then these affected components are scheduled using a parallel scheduler based on a straightforward variant of the standard topological sort algorithm.

The remainder of this chapter considers parallel *incremental* Makefile evaluation. We have already seen how lexical and syntactic analysis can be performed, but incremental construction of the symbol table and dependency graph is more complex. Once the symbol table, and the dependency graph represented therein, is constructed, it is straightforward to identify affected components using a transitive closure and schedule reevaluation based on a modified topological sort algorithm.

3.6 Incremental update of Makefile symbol table

From a practical point of view, there is little to be gained from incrementally processing Makefiles because they are typically small with respect to the size of the application

described. However, the techniques developed in this chapter will be formalized in the framework of attribute grammars and directly applied in the next chapter to the semantic analysis of more general programming languages. In short, these techniques present an introduction to and example of algorithms for the parallel incremental semantic analysis of more general programming languages.

The key to incremental parallel evaluation of a makefile is the maintenance of the symbol table, because the symbol table contains an explicit representation of the dependency graph. The symbol table may be updated by systematically identifying the parts of the Makefile program that have been deleted, inserted, and otherwise edited.

Our strategy for error detection and recovery is an integral and necessary part of the updating algorithm. It is necessary to have a consistent internal representation for even erroneous programs so that, when the user corrects the error, there is a meaningful 'previous symbol table' state from which incremental update may take place. We allow the symbol table to contain a number of definitions for each software component, so that there is a consistent symbol table state for the two classes of errors, undefined and multiply defined symbols.

The two program trees are recursively traversed simultaneously, subtrees in parallel. If two subtrees are pointers to the same memory location, then they are identical; the subtree was borrowed from the old version of the parse tree. Otherwise, structurally identical nonterminal nodes are considered a *change*, and their children are recursively compared. Structurally different nonterminals are treated as an insertion/deletion pair. Terminals are either identical or changed, depending on their value.

In the case of the makefile semantics, deleted rules are simply removed from the definition list of the symbol. If this leaves an undefined reference (that is, if the dependent list is nonempty), then the symbol is added to the list of undefined symbols. If

this leaves exactly one definition for the symbol, then the symbol is removed from the multiple definition list, if necessary. The symbol table entry is not unlocked until the undefined and multiply defined lists have been modified.

Inserted rules have their definition added to the definition list of the symbol, adding the symbol to the symbol table if necessary. If there are multiple definitions for the symbol, it is added to the multiply defined symbol list, if not already there. If it is the only definition, it is removed from the undefined list if necessary.

Changed rules are more interesting. These are rules that have been neither deleted nor inserted into the makefile, but whose description has been edited. We want to neither delete nor insert them into the symbol table[1]. Instead, we want to evaluate the changes in the rule, and update the dependency graph accordingly, thus processing only what has changed.

We may recursively apply this strategy to determine the difference between the prerequisite lists in a rule that has been edited. For each deleted prerequisite, the name of the component being defined is removed from the dependent list. Similarly, each inserted prerequisite has the current component added to its prerequisite list.

There is a catch: the rule list and the prerequisite list, both parts of the program tree, were parsed using list productions and are represented using balanced binary trees. Two trees representing identical syntactic lists might be represented by vastly differing parse trees! The straightforward technique described above, comparing trees node-by-node based on nonterminal type and terminal value, simply cannot cope with two semantically identical lists represented by different balanced trees. A specialized list comparison technique is needed.

[1] Actually, in the case of Makefile semantics, we could treat a change as both an insertion and deletion. For the purpose of more general programming language semantics, where large program parts nest, this simple strategy would require the unnecessary processing of large portions of the program that might not have changed.

The lists appearing in a makefile programs have the useful property that they are order insensitive. A set of rules given in any order defines the same dependency graph. Similarly, there is no semantic meaning in the order in a list of prerequisites presented in a rule. Thus, it suffices to compare the set of items in a list, independent of the list structure. Because programming languages that allow forward references have order independence in their declaration list, the techniques developed here will be useful there as well.

To enumerate the differences between the sets of items in two balanced list representations, we have developed a specialized list comparison algorithm that is described in detail in the next section. The algorithm compares two *keyed* lists represented as balanced trees and which may share structure, and enumerates inserted, deleted, and changed nodes; changed nodes are defined as those nodes with the same *key*, but with differing parse trees. The key, for the purposes of a makefile, is the component name in a component list or the name of a prerequisite in the list of prerequisites.

3.7 List Comparison

We describe a list comparison algorithm in which two syntactic lists, represented as balanced binary parse trees, and containing *keys*[2] at each item in the list, are compared. The previous and current versions of a list's parse tree are contrasted to determine which elements were inserted, deleted, or otherwise edited/changed in a list. Since the parser constructs the parse tree in an applicative manner, both the old and new parse trees are available to this algorithm. To support this tree comparison, list nodes of the parse tree are augmented with a tag field which is filled with a sequence number when the node is created during parsing. Each invocation of the compiler places the current sequence

[2]These keys are used to index a search structure, and for our application are the name of the target in a rule in a rule list or the name in a prerequisite list.

number in all newly constructed syntactic list nodes (nodes for nonterminals that have a list definition as introduced in the previous chapter) during parsing. The sequence number is incremented between compilations.

These sequence numbers can be used to discover that a large sublist of the syntactic list has not changed. If a node in the list's balanced tree representation has a sequence number that is less than the current sequence number, then the node and all of its children are borrowed from the old parse tree. This fact can be used to limit the number of elements of the list that are examined during the list comparison process.

The list comparison takes as input the old and new parse trees representing a list construct, and procedures for processing deleted, inserted, and changed elements of the list. In addition, it uses the *name* attribute from each list element as a key. The name is used to determine when the deletion of one element and the insertion of another is to be considered a *change* rather than independent operation; the rule we use is that this pair of operations is a *change* if the names are the same.

The tree comparison uses two auxiliary (concurrent linear) hash tables. The *sublist table* will be used to discover which list elements were deleted from the previous version of the list. The *name table* will be used to distinguish between elements that are new and those that were edited.

First, the data structure for a parse tree representing a list is assumed to be represented as in figure 3.7. Note that items in the list contain a **name** field which is used by the list comparison algorithm to determine when an insertion and a deletion together form a change pair.

The algorithm proceeds in three stages, first identifying what is on the new tree, then comparing it with what is on the old tree, and finally enumerating the results of the comparison in parallel.

```
type
    (* this illustrates tree comparison code corresponding to          *)
    (* The productions List := List List | Item                        *)
    tree_node = record
        sequence: integer;  (* seq. num when tree node generated *)
        case node_type: * of
        list =>
            (* nonterminal attributes *)
            case production: * of
                first =>    list1, list2: ^tree_node;
                second =>   item: ^tree_node;
                end;
        item =>
            name: string;        (* key for list comparison *)
            (* additional subtrees and attributes *)

        (* additional variants *)
        end;

var
    current_seq: integer;  (* curr. seq number for tree nodes *)
```

Figure 3.3: Data Structure for Tree Comparison

```
tree_compare(
    old, new: pointer to list tree_node,
    inserted, deleted: procedure(pointer to item tree_node),
    changed: procedure(old, new: pointer to item tree_node))
begin
    sublist_table: concurrent search structure = empty;
    name_table: concurrent search structure = empty;

    step1(new, sublist_table, name_table);

    step2(old, sublist_table, name_table, deleted, changed);

    step3(sublist_table, name_table, inserted);

    end;
```

Figure 3.4: Tree List Comparison Algorithm

```
procedure step1(
    new: pointer to tree_node,
    var sublist_table, name_table: concurrent search structure)
begin
    case (new^.node_type) of
        list => begin
            if (new^.sequence = current_seq) then cobegin
                step1(new.list1, sublist_table, name_table);
                step1(new.list2, sublist_table, name_table);
                end
            else sublist_table.add(new, new);
            end;
        item => begin
            if (new^.sequence = current_seq)
                then name_table.add(new^.name, new)
            else sublist_table.add(new, new);
            end;
        end;
    end;
```

Figure 3.5: Tree Comparison Algorithm: step 1

The tree comparison algorithm begins by enumerating the nodes of the new list that were created during the most recent reparse. These nodes are the ones that are either new or reflect changed source text. This first pass fills the **name_table** search structure with the new leaves of the list, keyed by name. It also constructs a *sublist table* that contains the topmost node of each subtree that is entirely borrowed from the old version of the list.

When the first step ends, the sublist table contains all nodes in the new list whose sequence number is less than the current one and none of whose ancestors have old sequence numbers. These are the nodes at the root of borrowed subtrees. The name table will contain all items in the list that are not borrowed from the previous version of the parse tree.

In the second step, the nodes of the old list are enumerated to determine which

```
procedure step2(
    old: pointer to tree_node,
    var sublist_table, name_table: concurrent search structure,
    changed, deleted: procedure(pointer to item tree_node))
begin
    if sublist_table.lookup(old) then begin
        sublist_table.delete(old);
        return;
        end
    else case old^.node_type of
        list => cobegin
            step2(old^.list1, sublist_table, deleted);
            step2(old^.list2, sublist_table, deleted);
            end;
        item => begin
            new: pointer to tree node = name_table.lookup(old^.name);
            if new != NIL then begin
                changed(old, new);
                name_table.delete(old^.name);
                end
            else deleted(old);
            end
        end
    end;
```

Figure 3.6: Tree Comparison Algorithm: step 2

of them were actually in the old list. This is necessary because old nodes were not necessarily in the same list in the old version of the program. It is possible for a node to have an old sequence value and yet not be found in the old list. This is the case when the user edits the boundaries between lists appearing in the target program. This phase detects and correctly deals with this possibility.

Sublists that were in common between the old and new lists need not be processed further. Leaves of the old tree, none of whose ancestors appear in the new tree, are processed as deleted or changed if their names also appear in new items, as found in the name table.

At the end of this step of list comparison, the sublist table contains exactly those subtrees that were in the new but not old list, but were from the old tree. These will be treated as insertions in step 3. The name table contains items whose names did not appear in the old list. Items whose names were in the old and new list have been processed as changed, and all deletions are processed here.

The third and final step of list comparison processes all other nodes of the new tree as inserted. These are the remaining items in the sublist table, which are borrowed from the old subtree but not the old list, and items in the name table, which are items with names that did not appear in the old list.

3.8 Representing the Makefile symbol table

The makefile symbol table is a search structure that represents the naming environment for the objects described in a makefile program. Since the makefile language supports only a single global scope, a single concurrent search structure suffices.

Each entry is keyed by the name of its component and contains a list of its definitions. There may be more (or less) than one definition in cases of an erroneous makefile; the

```
procedure insert_leaves(
    node: pointer to tree_node,
    inserted: procedure(pointer to item tree_node))
begin
    case node^.node_type of
        list => cobegin
            insert_leaves(node^.list1, inserted);
            insert_leaves(node^.list2, inserted);
            end;
        item => inserted(node);
        end;
    end;

procedure step3(
    sublist_table, name_table: concurrent search structure,
    inserted: procedure(pointer to item tree_node))
begin
    cobegin
        foreach node: tree_node on sublist_table do in parallel
            insert_leaves(node, inserted);
            end;

        foreach node: tree_node on name_table do in parallel
    insert_leaves(node, inserted);
            end
        end
    end;
```

Figure 3.7: Tree Comparison Algorithm: step 3

list allows us to incrementally handle this case gracefully. Each entry also contains a concurrent search structure to represent the set of prerequisites for the component.

Finally, a flag on each entry is used to reflect the fact that a component is out of date. These flags are set by transitive closure from changed components or rules, and only flagged items are scheduled for reevaluation.

To support error reporting and recovery, there is also a global table of multiply-defined and one of undefined components. These are updated when the symbol table is updated, and evaluation of the dependency graph will only proceed when both of these lists are empty.

3.9 Error Recovery

Since we maintain a list of undefined symbols, and a list of multiply defined symbol, error reporting is simply a matter of traversing these lists.

Error recovery is similarly straightforward. The symbol table contains a *list* of definitions for each component, one element for each definition seen. This provides a meaningful internal state of the symbol table for semantically erroneous Makefiles. When the user corrects these errors, each component will have exactly one definition. Thus, symbol tables even for semantically erroneous Makefiles may be used as the previous version during updates.

3.10 Scheduling the dataflow graph

Scheduling evaluation of the attribute graph can be performed using a straightforward parallel analog to the standard topological sorting algorithm. Once all file dates are checked, each component that is older than any of its immediate prerequisites is marked. Then, using a parallel transitive closure, components that depend on these (directly or

indirectly) are marked. This determines the set of components that need to be reconstructed. Each of these components is tagged with a *dependency count*, the number of marked values on which it depends. A parallel queue is maintained of components ready to be remade; initially this queue contains "leaf" components that can be remade immediately. Processors take tasks from this queue and begin processing them in parallel. When a processor completes updating a component, it decrements the dependency count of any components that depend on the one just completed. If this count reaches zero for other components, they are added to the ready queue, because the components on which they depend are all up-to-date.

3.11 Conclusions

We have described a strategy for incrementally maintaining and "compiling" a makefile program, based on the use of a specialized list comparison algorithm for the single scope's declaration list. In the next chapter, we generalize this technique and provide a notation to express it formally within the context of an attribute grammar, where it may be used to describe the semantics of more general programming languages. We also extend the techniques of this chapter to describe how to incrementally update the symbol dependency graph inherent in the symbol take for a program in a language described by a grammar that uses these extensions. As in this chapter, we will relate the parallel complexity of the algorithms to features of the language and target program.

4 Semantics and Code Generation

We wish to do the same for semantic analysis that we did for lexical and syntactic analysis: begin with a formal model that has been used in the generation of efficient sequential compilers and derive a parallel implementation by modifying existing methods. However, the state of the art in formal methods for semantics is not sufficiently advanced to admit translation from formal descriptions to a compiler that generates efficient object programs, even in the sequential non-incremental case. Developing such a formal model and method would be very valuable, but is beyond the scope of this thesis.

We will begin with *attribute grammars*, a semi-formal model for semantics that has been widely studied as a method for specifying compilers, and show how the method may be modified to support efficient parallel incremental semantic analysis. Without modification, attribute grammars require linear time to process the attributes representing the symbol table when only a single declaration is changed. We describe an extension to the formalism, and algorithms for implementing this extension, that allow sublinear time parallel incremental processing.

The time it takes to evaluate an attribute grammar in parallel is directly proportional to the depth of the attribute dependency graph. Since the semantics of a program depend (in general) on all of the nodes of its parse tree, and attributes are purely local, a tree of linear depth necessarily generates an attribute dependency graph of linear

depth. Conventional techniques for attribute grammars generate linear depth attribute grammars, and this problem has not been previously addressed. The main purpose of this chapter is to identify and eliminate sources of linear depth dependency chains. We show how the list productions introduced in the syntactic analysis chapter may be used to define attribute grammars to solve many problems in code generation and semantic analysis in parallel sublinear time.

Symbol tables and scope analysis pose a more difficult problem for parallel incremental semantic analysis. The simplest way to represent the symbol table in an attribute grammar is as an aggregate attribute that is passed around and modified at each declaration. Even hand-coded semantic analysis phases, implemented outside a formal system such as attribute grammars, are usually implemented this way. This technique is inappropriate in a parallel compiler because it creates a sequential dependency between a linear number of slightly different aggregates.

We separate scope analysis into three distinct phases. The first phase computes the referencing environments, determining which identifier definitions are visible in which regions of the source file. It does *not* evaluate the definitions of the symbols, because doing so would require it to refer to the definitions of other identifiers, whose relevant definition point is not yet known[1]. The output of this phase is a data structure that can be used to easily find the appropriate definition for a given identifier at a given location in the parse tree. The later phases use this data structure to coordinate the linking of identifier references with their definitions.

This chapter also discusses techniques for *code generation*, by which we mean instruction selection, variable and temporary allocation, assembly, and linking. Because these tasks are so dependent on the target architecture and choice of intermediate representation, we only discuss techniques that are widely applicable to a variety of

[1]This is what Seshadri et.al. [1988] describe as the DKY or *doesn't know yet* problem.

architectures.

4.1 Previous Work

Semantic analysis is perhaps the least understood facet of compiler theory. Lexical and syntactic analysis have a long history of widely accepted formal methods and a basis in automata theory that we have been able to adapt to the unique requirements of a parallel incremental compiler. On the other hand, there is no consensus among the research community of a single formalism for describing the semantic structure of programming languages for compiler construction. Denotational semantics, the most general and formal method for describing programming languages, is not yet useful as a basis for efficient compiler development. Operational semantics impose on the primitives an assumed knowledge of the features of the programming language being described or details of an underlying interpreter, imply a particular method for implementation, or only describe part of the semantics of a language.

4.1.1 Incremental Semantics

Among formalisms for semantics, attribute grammars are most commonly used because of the ease with which they are translated into executable compilers. This formalism is a result of the functional (side-effect free) values and the association of the computation with the structure of the parse tree. Their ease of implementation is a direct result of the fact that the description is algorithmic rather than declarative.

Attribute grammars are more of a programming language, or a way of organizing computations, rather than a formal method for describing the semantics of a language. It is a framework in which arbitrary values, or *attributes*, are associated with nodes of a parse tree. Each attribute is described as a function of attributes of neighboring nodes. Systems that use attribute grammars typically require that the dependency

graph among attributes is noncircular, so the attribute values may be computed by a topological evaluation of the dependency graph. Other, stronger restrictions on an attribute grammar may allow slightly more efficient evaluation algorithms.

Unfortunately, a simple attribute grammar description of the scoping structure of a language is usually inefficient for incremental semantics. The two main problems are long chains of copy values and the representation of aggregate values for symbol tables. Both of these must be addressed to achieve reasonably efficient incremental evaluation.

There have been two basic strategies for dealing with the problem of long chains of attribute copies. First, the implementation of the attribute grammar formalism can maintain *shortcuts* for the copy chains automatically [Reps, 1982; Hoover and Teitelbaum, 1986]. This solution relieves the compiler writer from being concerned with the details of the solution.

The second approach is *aggregates*, which extends the attribute grammar formalism by providing the compiler writer with a way to express relationships between attributes that are not adjacent in the tree [Johnson and Fischer, 1982; Johnson, 1984; Beshers and Campbell, 1985; Reps *et al.*, 1986; Hoover and Teitelbaum, 1986]. An aggregate is a collections of attributes that may be passed between tree nodes as an attribute, but which contains a number of attributes that can be individually referenced. This method can allow a more natural description of the programming language because an aggregate can be used to represent the symbol table for a scope.

Various techniques have been used to improve the efficiency of incremental attribute grammar evaluation for copy and aggregate values [Fischer *et al.*, 1984; Hoover, 1987], but these methods do not support the arbitrary structural changes to the parse tree that are possible with a conventional text editor. The approach of combining incremental parsing based upon a text paradigm with incremental evaluation of the attributes within

the reparsed program is new. We borrow many concepts and ideas, but few algorithms from these researchers.

4.1.2 Incremental Code Generation

In some incremental environments, code generation is performed in the background or on demand, when the code is first executed [Delisle *et al.*, 1984]. Although some systems will generate code only for changed statements [Fritzson, 1984] most regenerate code for an entire changed procedure [Delisle *et al.*, 1984; Schwartz *et al.*, 1984]. Many environments avoid the code generation issue by interpreting the program [Teitelbaum *et al.*, 1981].

The appropriate granularity for incremental code generation depends, among other things, on the environment. Delisle *et al.* [1984] consider the problem of a target program that is running on a physically remote machine. Because of the cost of changing the program, minimal "patches" are sent when the program is changed. In this system, the grain of incrementality is the statement, which incurs a large cost in space because the code for each statement (and for blocks of statements) must be kept; in addition, there is significant processing involved in patching new code into old. For instance, when a variable declaration is deleted, it leaves a "hole" in the data area, which can be filled later by newly declared variables. Similarly, if a statement is inserted into a section of code where there is not enough room for the new code, then the code is placed elsewhere, and a branch to (and back from) the misplaced code must be inserted.

The smallest possible granularity for incremental code generation is the expression, but the smallest grain that has been used in the literature is the statement [Fritzson, 1982]. The reason is that, even when the selection of instructions has not changed, all of the code after a changed statement must be moved to adjust to fit the size of the new code for a statement. This is a result of a common design decision that the code for a

procedure should be contiguous.

Most implementations generate code for an entire procedure when part of the procedure has changed. The code generation is much simpler, and therefore potentially faster. Also, this simplifies the earlier portions of the compiler by requiring incremental processing of units no smaller than the procedure.

4.1.3 Parallel Semantics

An event-driven model is one technique for parallelizing semantics. In this model, the availability of a semantic value triggers the computation of other values [Andre *et al.*, 1981; Banatre *et al.*, 1979; Lipkie, 1979]. This is closely related to the use of Petri Nets by Baer and Ellis [1977]. Although the intent of these projects was to help design and understand sequential compilers, these techniques have been shown to extract a useful amount of parallelism. However, the available parallelism in semantic analysis using these techniques is limited by a small constant, because symbol table maintenance is not parallel.

Researchers at the University of Toronto [Seshadri *et al.*, 1987; Seshadri *et al.*, 1988] are taking a more wholistic approach to concurrency in compilation. In their model, the program is divided up along natural boundaries in the program text, such as procedures or modules. Each section is then compiled through all phases by a devoted processor. Symbol table processing, a necessarily nonlocal computation, is the most communication-intensive of the compilation tasks and proceeds by the use of a shared data structure. The processors communicate to construct and use this shared symbol table. When a processor looks up an identifier in an enclosing context and it is not found, it might be because the enclosing context has not been fully processed. In this case, we do not know if we should search a more global scope or not. This is described as the "doesn't know yet" problem, for which the authors present a number of alternative

solutions. Their basic solution is to wait for the needed scope to be completed, after which the identifier can be looked up in that scope. If the definition is not found, it is searched for in a more global scope. When an appropriate definition is finally found for the referenced identifier, semantic processing can proceed.

There are two main drawbacks to this solution. First, the processor is not doing useful work while it is waiting. The authors address this issue in a refinement to their solution by allowing the processor to work on other semantic processing in its segment of the program. While this will improve processor utilization and concurrency, it is at the expense of increased synchronization overhead and memory utilization. Furthermore, the processor may still run out of useful work to perform while waiting for a definition, leaving the processor idle, perhaps because of a poorly balanced subdivision of the program at the highest level.

The second major drawback is that a processor may wait for a scope to complete processing, even though the needed definition is from a more global scope. In other words, because the processor "doesn't know yet" which scope the identifier is declared in, it must wait for processing to complete in any possible defining environment. This introduces into the semantic processing an unnecessary delay and synchronization. Worse, a problem that the authors do not consider is a circular dependency between scopes. A scope that both imports one identifier and exports another will cause a circularity between scopes. In this case, there is no ordering of processing scopes that will avoid deadlock. Seshadri et.al. address the performance problem in a way that would solve the deadlock problem, by considering an "Optimizing Symbol Table Search," in which semantic analysis proceeds in two separate phases. The first phase, processed during parsing, records a list of the defining scopes for each distinct identifier appearing in the program, represented as a bit vector for efficiency. Then, during the second phase of semantic processing, this list can be intersected (bitwise and-ed) with the list of parent

scopes to determine the correct defining scope for an identifier. The processor must still wait for the identifier's relevant definition to be processed before proceeding with the semantic analysis.

This optimization has the advantage that it will allow higher utilization of the processors, but as we shall see it is at the expense of more work. Unfortunately, the execution time of semantics using the "optimizing" table search is asymptotically no better than (that is, within a constant factor of) a sequential implementation.

Consider the ideal situation for Seshadri's compiler, in which a program of length n is divided into p procedures or modules of identical length; the program is divided along these boundaries for processing. Each identifier appearing in the program will have a bit vector associated with it of length p. Let us also assume that there are $\Omega(n/p)$ identifiers appearing in each module (the number of identifiers being proportional to the length of the text). Now, to process each identifier a string of p bits must be masked and searched. This will take time $\Omega(p)$ for each identifier, and time $\Omega(pn/p)$ or $\Omega(n)$ for one entire module. Though the constant factor may help, this is no asymptotic time improvement over uniprocessor compilation. The work done, $P * T = \Omega(np)$, is proportional to the number of processors used. We hope to do much better for such well-balanced programs.

Attribute grammars have been used in the description of languages for the implementation of parallel semantic analysis [Boehm and Zwaenopoel, 1987]. Attribute grammars are used because of the ease with which they are translated into executable compilers, and the applicative values lend themselves well to parallel evaluation because of the absence of side-effects.

Unfortunately, a conventional attribute grammar description of the scoping structure of a language is inefficient for parallel semantics. Declarations are processed in the

strictly sequential order imposed by the attribute grammar techniques. Although the parallel attribute evaluator of Boehm and Zwaenopoel [1987] was able to achieve an effective speedup for a small number of processors, their data show declaration processing to impose a sequential constraint on semantic analysis. We modify the attribute grammar formalism in a way that allows the compiler to achieve effective speedup (that is, more than a constant factor) even when the declarations are a constant fraction of the program text.

Because the computations specified by the attribute functions are arbitrary, we cannot hope to parallelize them individually in any automatic way. While this might be an important source of practical parallelism in the compiler, each of these functions would have to be parallelized using a technique specific to the function being computed. This chapter considers several classes of attributes that are commonly used to implement compilers, and describes techniques that improve the parallelism of their computation by organizing the attribute grammar to allow maximal overlap between evaluation of different attributes.

4.1.4 Parallel Code Generation

Many researchers have been able to successfully exploit parallelism in code generation. *Baer et. al.* [1977] have shown that there is a great deal of available parallelism in code generation. *Vandevoorde* [1988] had success with code generation; his compiler was limited mainly by symbol table analysis for the small number of processors he had available. One of his important contributions is `WorkCrews` [1988], a programming library based upon an abstraction that automatically clusters tasks in divide-and-conquer style parallel algorithms with very small overhead.

4.2 Attribute Grammars

The research to date on using attribute grammars for parallel semantics show that there are two major sources of sequential constraints: tree depth (as a result of long lists in the source program) and symbol tables. We extend the attribute grammar formalism to alleviate the two problems.

4.2.1 List Productions

Attributes can be associated with list productions just as for other productions. The balanced form of the parse tree (and the desire to limit the depth of the attribute dependency graph) implies a different paradigm for writing the attribute grammars. Although the parse tree is ambiguous, the compiler writer wants the semantics defined by the compiler, and expressed in the attribute grammar, to be unambiguous. We describe two standard techniques for computing attributes of a list represented by a balanced tree, and show how they may be used for problems occuring in semantic analysis and code generation.

Our model of parsing complicates any incremental attribute grammar scheme. Normally, an unambiguous grammar is used to parse the program, so the generated parse tree is unique. In this case the amount of change made to the structure of the source program corresponds to the amount of change to the parse tree. However, we have intentionally introduced an ambiguous grammar so that lists may be represented as balanced binary trees. While this eliminates one source of deep attribute graphs (deep parse trees), the structure of the parse tree, and therefore the structure of its underlying attribute graph, is not uniquely determined.

List productions appear in the syntactic grammar as follows:

```
List ::= List List
```

This special form is recognized by the parser as a special case of interest for list productions, so that the list is may be restructured and represented as a balanced tree. It is also directly useful for specifying attributes because attributes for this production may be defined just as for any other production in the grammar. We allow both synthesized and inherited attributes to be defined for list productions just as for normal productions in the grammar:

```
List ::= List List {
    List[0].synth = func (List[1].attr, List[2].attr);
    List[1].inh = func (List[0].attr); }
```

The usual deeply nested tree representation of list structures, such as declaration and statement lists, is one source of long attribute dependency chains in an attribute grammar. In modifying the parsing formalism for list structures, we have made it possible for the compiler write to solve this problem. However, because the grammar is ambiguous, special attention is required from the compiler writer to construct an unambiguous description of the language semantics.

Using List Productions

We describe two techniques for parallel algorithm design that can be expressed directly in the attribute grammar using the list productions.

The first technique is divide-and-conquer. If a large problem can be divided into two nearly equal size subproblems that can be solved in parallel, and whose results can be combined quickly, a viable parallel algorithm results. Because the lists are represented as *balanced* trees, their structure provides a natural vehicle for this technique. Static semantic analysis and instruction selection[2] of independent declarations, statements and subexpressions cleanly fit this model. Divide-and-conquer algorithms can be expressed

[2]But not the computation of *where* to put the instructions.

directly in the attribute grammar and are well suited to a formal complexity analysis. For example, if combining subproblem results requires $O(\log n)$ parallel time, and the subproblems at the leaf items can be solved in $O(\log n)$ time, then divide-and-conquer requires only $O(\log^2 n)$ time and $O(n)$ processors.

Parallel algorithms based on the technique of *parallel prefix* [Ladner and Fischer, 1980] are useful for a number of problems arising in code generation, and can also be expressed directly in an attribute grammar. The simplest form of parallel prefix algorithms[3] can be used to allocate and assign memory locations to instructions and data, to compute the size and layout of aggregate data types, and to compute the 'next' code location for each statement in a list.

Parallel prefix can also be used to detect attempts at forward identifier references in static semantic analysis, by computing the number of tokens preceding each declaration tree node. A forward reference is detected when there are more tokens preceding the definition than preceding the use. Combined with algorithms for scope analysis based on *upward remote aggregates* that do not detect erroneous forward references, an efficient parallel static semantic analysis is possible.

Code Generation

Code generation is not a focus of the dissertation, mainly because the problem is not too difficult: code for separate statements, basic blocks, and procedures can be generated nearly independently, thus proving ideal for parallelism. Existing techniques for incremental code generation similarly parallelize in a fairly straightforward way.

Code generation by coagulation [Karr, 1984] is a technique for bottom-up instruction selection that can be expressed as a divide-and-conquer style computation, and is therefore ideal for a parallel implementation. Graham-Glanville code generators are based

[3] Using the associative integer addition operation.

on parsing technology for instruction selection, and our parsing techniques could be applied to these algorithms as well to provide very fine-grained parallel code generation.

Perhaps the biggest obstacle to incremental code generation at a fine grain is the requirement that the code for a procedure be contiguous. This requirement is rarely a result of a conscious design decision by the compiler developer, but a result of a historical bias from sequential compilers, in which it is easiest to generate continuous code. However, changing a small section of code near the beginning of a large procedure should not require the later sections of the procedure to be reprocessed merely because the early statements require more memory space. For the same reasons, this can be a difficulty with parallel code generation as well. While techniques such as parallel prefix can be used to compute the appropriate code positions after instructions have been selected to cause a procedure to fit into contiguous memory, these techniques are not appropriate in an incremental context because they preserve few old code positions.

We see the code location problem, and the related data location problem, as special cases of *parallel memory allocation.* The problems are special cases only because known memory size distributions on basic blocks could be used to design an especially efficient allocator for the problem. Indeed, a parallel memory allocator is a crucial part of the library for a shared-memory application environment, and a number of researchers have investigated appropriate algorithms. We do not survey their results, but only note that their results are directly applicable to this problem.

4.2.2 Symbol Tables: Upward Remote Aggregates

Long path lengths are also generated by the description of symbol tables. Symbol tables introduce two different types of long paths in the attribute graphs. First, a symbol table itself is passed throughout an entire scope, so it may be used everywhere. The obvious method for passing symbol tables throughout a scope, by copying the symbol table

attribute to each node, imposes a linear depth dependency subgraph; this is due to the linear depth with which lists of declarations are represented. Second, the symbol table is typically built in an inherently sequential way for a single scope, adding each declaration to a symbol table to create a slightly different referencing environment for the next declaration. A symbol table described in this way generates a linear depth dependency graph, in which declarations are added one by one to create a linear succession of symbol tables.

Most implementations of scope in compilers create a different version of the symbol table after each declaration. This appears to be necessary in languages such as Pascal, where each declaration introduces a new *contour* [Garrison, 1987], or referencing environment. Furthermore, these symbol tables are sequentially computed, one after the other in the order that the declarations appear in the source. This seemingly inherent sequentiality of symbol table maintenance appears in both ad-hoc and formal methods for semantics.

The attribute grammar in Figure 4.1 illustrates the essence of these two key problems. The only potential parallelism in this symbol table maintenance scheme is that different non-nested scopes and statements in different scopes may be processed simultaneously.

Without modification, this attribute grammar requires linear time to process the attributes representing the symbol table when only a single declaration is changed. The reason is that the parse tree is of linear depth for a list of declarations, and the attribute dependency graph is therefore also of linear depth.

The deep dependency graph is a serious impediment to parallel semantic analysis. A naive evaluation of the attribute grammar must traverse the depth of the dependency graph sequentially. A linear depth dependency graph imposes a severe sequential con-

```
Program
        :       Block
                        { Block.env_in := nil }
Block
        :       "BEGIN" DeclList "DO" StmtList "END"
                        { DeclList.env_in := Block.env_in;
                          StmtList.env := DeclList.env_out }
DeclList
        :       <empty>
                        { DeclList.env_out := DeclList.env_in }
        |       DeclList Decl
                        { DeclList[1].env_in := DeclList[0].env_in;
                          Decl.env := DeclList[1].env_out;
                          DeclList[0].env_out := Decl.env_out }
StmtList
        :       StmtList Stmt
                        { StmtList[1].env := StmtList[0].env;
                          Stmt.env := StmtList[0].env }
        |       <empty>
Decl
        :       id ":" id
                        { CheckUse(id[2]);
                          Decl.env_out := AddDecl(
                                Decl.env, id[1],
                                ExtractDef(id[2], Decl.env)) }
Stmt
        :       id
                        { ExtractDef(id, Stmt.env) }
        |       Block
                        { Block.env_in := Stmt.env }
```

Figure 4.1: Simplified Grammar for Pascal Scopes

```
Program
        :       Block
                        { Block.env_in := nil }
Block
        :       "BEGIN" DeclList "DO" StmtList "END"
                        { DeclList.env_in := Block.env_in;
                          StmtList.env := DeclList.env_out }
DeclList
        :       Decl
                        { Decl.env := DeclList.env_in;
                          DeclList.env_out := Decl.env_out }
        |       DeclList DeclList
                        { DeclList[1].env_in := DeclList[0].env_in;
                          DeclList[2].env_in := DeclList[1].env_out;
                          DeclList[0].env_out := DeclList[2].env_out }
StmtList
        :       StmtList StmtList
                        { StmtList[1].env := StmtList[0].env;
                          StmtList[2].env := StmtList[0].env }
        |       Stmt
                        { Stmt.env := StmtList.env }
Decl
        :       id ":" id
                        { CheckUse(id[2]);
                          Decl.env_out := AddDecl(
                                Decl.env, id[1],
                                ExtractDef(id[2], Decl.env)) }
Stmt
        :       id
                        { ExtractDef(id, Stmt.env) }
        |       Block
                        { Block.env_in := Stmt.env }
```

Figure 4.2: List Productions

straint on symbol table analysis, and therefore on the entire compilation; by Amdahl's law, this limits effective parallelism to some small constant.

Using list productions, we can express these semantics in a way that allows more parallelism. The attribute grammar formalism is not modified, except to use the usual method for describing attributes on the special list productions. With list productions, the depth of the parse tree is only logarithmic in the length of a declaration list.

This transformed grammar is superior in a crucial way: different statements in the same scope may be processed simultaneously. However, this grammar still suffers from a sequential processing of declarations. Even though the depth of the parse tree has been reduced to $O(\log n)$ for the declarations in a single scope, the attribute graph has not been reduced in depth.

The reason is that a symbol table is passed from declaration to declaration and a linear chain of different scopes are sequentially dependent. To evaluate the attributes as written, the symbol table must be passed from one declaration to the next for modification, in order. We can take some advantage of the structure of the problem space to improve the situation slightly without changing the formalism. The symbol table is actually modified in only one place:

```
Decl.env_out := AddDecl(
      Decl.env, id[1],
      ExtractDef(id[2], Decl.env))
```

However, this is equivalent to

```
Decl.env_out := MergeEnvironments(
      Decl.env,
      NewEnvironment(id[1], ExtractDef(id[2], Decl.env));
```

where NewEnvironment creates an environment with a single binding, and MergeEnvironments combines the bindings from two environments. Because MergeEnvironments is associative we can transform the grammar into one that uses parallel prefix to compute the environments. A grammar using parallel prefix describes an equivalent dependency graph among name-value bindings. To be evaluated directly would still require a sequential linear number of evaluations though, because each singleton environment constructed depends on all of the declarations to its left.

Consider what would happen if we allow the environments to be evaluated lazily; that is, the environment is constructed before the binding values of its identifier is computed. In this way, the environment is treated as a collection of attributes rather than a single attribute. This could be accomplished by placing a Multilisp-style *future* [Halstead, 1985] around the value computation where a singleton environment is created for a declaration, instead of waiting for the environment from the left to become available.

Because futures are functional, they do not influence the semantics of the computation. The advantage is that the set of bindings (with lazy values) could be constructed in parallel, because their interdependency does not constrain the computation of the referencing environment. After this is done, the only constraint to the parallelism is the actual dependency graph between declarations. In this way, the symbol table ceases to be a sequential constraint.

This lazy evaluation is one form of separation between dependency graph construction and dependency graph evaluation, and is the crucial insight into our algorithms for semantic analysis. This technique effectively separates determining where each identifier is defined (i.e., computing the naming environment) from determining what the value of its definition is and where it is used. It allows us to efficiently construct the environments before computing the values associated with the bindings. Such a separation is necessary for efficient parallel (and therefore *parallel incremental*) semantics.

If MergeEnvironments takes $O(1)$ time using $O(n)$ processors, then this environment can be computed in O(log n) time using $O(n^2)$ processors. (Merging environments with this expected time complexity can be accomplished using a hash table representation of the environments.)

The $O(n^2)$ processors utilized in the construction of the lazy environments is unacceptably large. Note that the symbol table for a language that allows arbitrary forward referencing, such as Modula-2, can easily be processed using a variant of this technique in $O(\log n)$ time using $O(n)$ processors. The reason is that the upward pass, merging environments, is more efficient than the downward pass, where many large environments are created. The downward pass is used to compute the contour after each declaration in a scope. Modula-2[4] does not need the downward pass, because a scope is a single referencing contour [Wirth, 1982].

[4]At least, its original definition.

This optimization, in the non-incremental case, can be extended for languages such as Pascal, where each declaration does introduce a new scope. The trick is to process the symbol table the same way as for Modula-2, creating a single contour containing all of the local declarations, and use a separate technique for detecting and preventing forward references. One such technique is to assign *sequence numbers* to all declarations in a scope, and emit an error message if the searched identifier has a higher sequence number than the current declaration (i.e. it is an attempted forward reference). These sequence numbers can be created in $O(\log n)$ time using $O(n/\log n)$ processors using the simplest version of parallel prefix; these computations can be expressed directly in the attribute grammar.

The incremental case is more difficult. First, forward references cannot be efficiently disallowed using sequence numbers, since this would cause a large number of sequence numbers to change when one early declaration changes. We do not attempt to solve this problem; languages generally disallow forward references because language design is based on the available compiler technology. We are introducing a new compiler technology that relaxes this language constraint.

This ambiguity in the parse tree representation of a list introduces the unfortunate possibility that two parse trees, while structurally different, represent the same program. After a small incremental change to a program, the parse tree representation may be drastically altered. Consider the case in which a user edits comments within a declaration list: the declaration list may be reparsed into a totally different balanced representation of the same declaration list. We efficiently handle this case by extending the attribute grammar formalism so it is less sensitive to the local structure of the parse tree, using a combination of the ideas of *upward remote references* [Reps *et al.*, 1986] and *aggregate values* [Hoover and Teitelbaum, 1986].

Languages that disallow forward referencing are much easier to implement as se-

quential, hand-crafted compilers. On the other hand, we have seen that languages that allow forward referencing are much easier to implement as parallel compilers. This is one reason to prefer forward referencing in language design on the basis of its impact on parallel compiler design.

We solve these problems in a more formal way by introducing *upward remote aggregate* attributes. These special attributes, appearing at the top of the subtree representing a scope, are used to represent the symbol table for that scope. These aggregates may be referenced anywhere within the scope without copying the entire symbol table to a local attribute. Because the aggregates may be referenced nonlocally, the long chains of copies of the symbol tables are avoided. Also, we allow the declarations to be added to the aggregate in any order, as well as simultaneously; this avoids the sequential constraint of processing declarations in order.

Specifying URAs

It is desirable to have a single representation for the dependency graph among attributes, rather than having some represented implicitly as futures, so that we may efficiently schedule based on the structure of the whole dependency graph. We therefore consider the referencing environment primitives as a special part of the attribute grammar formalism. Referencing environments, or scopes, can be constructed in one pass, without regard to the values of the bindings. Because the bindings in the definitions are not evaluated, the environments themselves can be constructed in parallel without regard to the interdependence of their values.

The primitives available in the attribute grammar are:

```
Aggregate := EMPTY;
```

This defines an aggregate value that contains only what is added to it through

remote upward references. The name on the left-hand side is the name of the scope. This primitive is used to create the outermost scope of an implementation module or the outer scope for a local module. To add something to the aggregate, the following form is used:

```
Add(up(Aggregate), Key, Value);
```

This declares the aggregate to contain, in addition to what may be added elsewhere, an association between 'Key' and 'Value'. This may appear anywhere in the subtree of the parse tree that contains the definition of the aggregate, but not within the subtree of another aggregate of the same name lower in the parse tree. Thus, the primitives implement the concept of nested scopes. We restrict 'Key' to be a string value computed by strict synthesis, so that it may be computed and maintained during parsing. In the description of languages such as Pascal and Modula-2, these keys appear as values in the leaf symbols for identifiers being defined.

Nested open scopes can be described by creating a new scope whose default definitions for identifiers are inherited from another scope.

```
Aggregate := Inherit(up(Aggregate));
```

These default definitions may be overridden by adding associations, as before. In a typical use of aggregates to represent symbol tables, the name of the remote aggregate is the same as for the local aggregate.

Values may be extracted from an aggregate by key. As before, the key is a string that must be available as a strictly synthesized value:

```
Value := Lookup(up(Aggregate), Key);
```

This retrieves the definition of *key* in the lowest ancestor in the parse tree that contains a definition for the named aggregate.

Unlike other definitions of aggregate values, our symbol tables are "stateless," do not imply any ordering of the execution of the primitives, and do not require user-defined retractions to implement incremental processing. On the other hand, our primitives are less general than those that allow implementor-defined retractions because our symbol table primitives are tuned to the needs of a particular language definition (the language described by our example grammars). Because the form of the primitives is so restricted, we are able to implement the retractions in the compiler-generating system rather than requiring their definition by the language implementor. However, we believe that these primitives can easily be extended to support a wider class of languages.

Using URAs

We have given a sample grammar for the scoping structure of Pascal. We now rewrite the grammar using *upward remote aggregates.* Note that there is one crucial difference between the semantics described by this grammar and the previous examples: *this grammar allows forward referencing.* The techniques we have introduced would only be complicated by the need to check for forward references to give error messages.

This is a difference between what is easy to implement in parallel and sequential compilers. Sequential compilers are much simpler if the language does not allow forward references, but just the opposite is true of parallel compilers. Just as language design has historically been influenced by the compiler technology available, this is an argument to allow forward referencing in languages whose compilers may be parallel. Since there is no reason other than convenience to the compiler writer (at the expense of the programmer) to disallow forward references in a compiler, we will concentrate our attention on a variant of the example language that allows forward references.

Using the URA primitives, the example grammar, now allowing forward references, can be written as in Figure 4.3.

```
Program
        :       Block
                        { LocalSymbols := EMPTY }
Block
        :       "BEGIN" DeclList "DO" StmtList "END"
                        { LocalSymbols := Inherit(up(LocalSymbols)) }
DeclList
        :       Decl
        |       DeclList DeclList
StmtList
        :       StmtList StmtList
        |       Stmt
Decl
        :       id ":" id
                        { Key := Id[1];
                          Type := Lookup(up(LocalSymbols), id[2];
                          Add(up(LocalSymbols), id[1], Type }
Stmt
        :       id
                        { Type := Lookup(up(LocalSymbols), id }
        |       Block
```

Figure 4.3: Sample Grammar using Upward Remote Aggregates

4.3 Implementing URAs

Without loss of generality, we consider only a single aggregate name appearing in the grammar, as in the sample grammar. This simplifies the exposition, and these algorithms easily extend to multiple aggregates. Since the grammar has a constant number of aggregates, the algorithms described here could be modified to process them simultaneously or sequentially.

The 'normal' attributes of the attribute grammar may be implemented in a conventional way, even though the grammar is ambiguous. However, the upward remote aggregates require special attention. Because the symbol dependency links are nonlocal, it is not possible to process them in a purely local way.

The basic strategy will be to maintain a search structure at each tree node for which the attribute grammar contains an aggregate definition. This structure will be constructed or updated in one phase to create a description of the referencing environments, and another phase will link references with definitions. By updating the data structure describing the set of available definitions at each point in the program before

we process the references, we avoid the problem of using symbol definitions before we know the correct scope in which to search for them.

Each symbol table entry is a special kind of attribute in the attribute grammar. Rather than being associated with a parse tree node, it is associated with an aggregate and a *key*, or name by which the search structure is indexed. Each entry is dependent on the attribute of its definition, and each reference to the identifier is dependent on it.

While the parse tree is traversed recursively and in parallel to update the symbol dependency links, the searching procedures maintain a pointer to the current (that is, lowest ancestor) aggregate so the appropriate search structure for the contour is immediately available. All identifier references are to entries in the local symbol table; even identifiers from an enclosing scope are entered on the local symbol table and placed on the dependency list of the definition in the outer scope. The aggregate attributes themselves (i.e., the symbol tables) do not appear in the constructed dependency graph, but are used only to coordinate the construction of the dependency graph among the symbols.

Semantic analysis proceeds in distinct phases: First, purely synthesized attributes from the attribute grammar are computed, providing attributes that will be used as *keys* in the list comparison algorithm and for aggregate element names. These attributes may be maintained during parsing. Next, the parse trees (old and new) are traversed and compared to enumerate changes to the scoping structure and set of definitions in the program, and the symbol tables are updated to reflect these changes. Following this, the symbol table correctly and completely reflects the identifiers defined in each contour. In addition, each attribute in the aggregate is linked to its defining attribute.

Then, the parse trees are once again compared, but this time to determine how the set of identifier references have changed. The attribute dependency list of each attribute

in the aggregate is updated to contain exactly the set of references to that definition that appear within that scope.

These latter two phases construct the inter-symbol dependency graph. Together with the attribute dependency graph, this forms a directed acyclic attribute dependency graph[5] that is subsequently scheduled for evaluation. This section is concerned primarily with the maintenance of the inter-symbol dependency graph.

4.3.1 Representing the Aggregate

Aggregate attributes in the attribute grammar are represented by a specialized data structure in the parse tree, described in pseudocode in Figure 4.4. These aggregates are not part of the attribute dependency graph in the usual sense, but are containers for the remote aggregates that are linked into the dependency graph. Each aggregate contains a pointer to the parent aggregate (*i.e.*, the aggregate higher in the parse tree with the same name), the type of relationship with the parent (*i.e.*, open or closed scope), and a search structure that contains the name-attribute pairs.

The aggregate attribute, or symbol table, for each contour contains a concurrent search structure for the names defined or used within the scope, and a pointer to the symbol table for the parent scope. Each symbol entry within the scope contains a list of dependent attributes.

Every symbol reference within the contour is treated as a reference to a symbol definition in the search structure for the contour. Thus, identifiers that are defined in the outer scope are entered in the local scope, and a dependency link is created between the local and parent entries. The only other type of dependency is the upward remote aggregate reference, which is a dependency attribute attached to a tree node.

[5]It is possible, given an erroneous program, to have a cyclic dependency graph. This problem is discussed in a later section on error detection and recovery.

```
type
    attribute_value = record
        (* place to store the attribute data *)
        prerequisites: concurrent list of ^attribute_value;
        dependents: concurrent list of ^attribute_value;
        case type from
            tree_node => node: ^parse_tree_node;
            symbol => table: ^scope_value,
                      entry: ^scope_entry;
        end;

    scope_entry = record
        lock: semaphore;
        name: string;
        value: attribute_value;
        end;

    scope_value = record
        parent: ^scope_value;
        scope_type: (open, closed);
        values: concurrent search structure of ^scope_entry;
        imports: set of ^scope_entry;
        children: set of ^scope_value;
        end;
```

Figure 4.4: Aggregate Data Structure

References to symbols from enclosing scopes, in the case of a scope that imports definitions from its parent, are treated specially. An entry is placed in every scope between the defining scope and the referencing scope. Each of these *import* entries is an attribute in the dependency graph that depends on the parent's definition and on which references within the scope depend. Thus, all references within a scope go through the local symbol table.

To support incremental updates, we store in the aggregate a list of the imported symbols. This list contains a pointer to the attribute in the local symbol table for each imported symbol that was referenced within the open scope. The import list for a closed scope is, by definition, empty. When a scope is deleted, this list will be used to avoid traversing the entire scope to delete references to the parent scope.

4.3.2 Updating the Symbol Table

The symbol dependency links are maintained in two phases. The first updates the links between definitions and their symbol table entries. The second updates links between symbol references and the definition attribute in the aggregate. This section describes the first of these two phases.

The symbol table is updated during a parallel recursive comparison and enumeration of the parse trees, using the algorithm described in the previous chapter. Where the grammar writer has provided purely synthesized `Key` attributes, the algorithm will use these keys to determine what nodes are to be considered changed. This is primarily useful to the compiler writer in declaration lists to allow tree comparison to identify changed procedures, and allows the procedure body and nested declarations to be processed incrementally.

The recursive comparison procedures are modified to maintain, as a parameter, a reference to the lowest enclosing aggregate. Thus, the search structure in which

all definitions should appear is immediately available. When the symbol tables are referenced, it is sometimes necessary to modify the enclosing scope. Modifications are needed in the parent scope during this phase because the insertion of a definition can override an imported definition, and the dependency link with the imported definition needs to be removed.

When a symbol table entry is being modified, it is locked using a semaphore associated with the entry. To prevent deadlock, locks are acquired from the innermost to the outermost scope being modified, in order. When it is necessary to modify the symbol table entry for the parent scope, the local scope's entry for that key is locked first. This rule also applies to the later phase that updates the references to each identifier.

Inserted and Deleted Scopes

When the tree comparison routines encounter a parse subtree that is new and whose root contains an aggregate attribute, a new aggregate data structure is created and linked with the parent scope. The contents (subtrees) of the scope are then traversed to enumerate all declarations and add them to the aggregate. Since the scope is new, there is no special processing required for imported identifiers; these will be processed as symbol references in the following phase.

During this traversal, any subtrees that are encountered that also have aggregate definitions are similarly handled, recursively, to create any new nested aggregates.

An entire scope is considered deleted when its *scope head*, the defining occurrence of the aggregate attribute in the old tree, is removed from the parse tree by the user's edits. Since the scope contains a list of imported identifiers, these are enumerated and the dependency link with the enclosing scope is removed for each one. Traversal is required no deeper in the parse tree, since this imports list localizes the interaction with

the upper portions of the parse tree. Once these imported definition links are removed, there are no remaining dependence links between the new parse tree and the deleted subtree.

When a reference is deleted to a definition from an enclosing scope because of the deletion of a scope that imports that definition, the enclosing scope's entry is locked and checked to see if it, too, is an imported definition and if this is the sole remaining reference to it. If this is the case, the imported definition is recursively removed from the grandparent's scope as well. This is done on up the tree until the source definition for the identifier is reached, in which case it remains in its local aggregate.

Changed Scope: Inserted and Deleted Declarations

Changed scopes are the most interesting. This is the case when the tree comparison routines discover that the tree node that contains an aggregate definition has been changed from the previous version of the parse tree. When this happens we need to identify the set of definitions from the scope that were deleted, inserted, and changed, and update the aggregates to reflect this information.

Newly inserted declarations are added to the definition list in the current scope. If there is already an imported definition, that import reference is removed from the enclosing scope and the imported definition is removed from the list of definitions for the symbol and from the imports list of the scope. As before, the removed import may be recursively removed from enclosing scope's aggregates.

A deleted definition is removed from the definition list of the key in the current scope. If it is the last definition for the key and there is a reference to the attribute on the list of its dependents, then it is automatically made a reference to a definition from the enclosing scope. If there is no definition in the enclosing scope, one is created

and imported from its enclosing scope. This repeats upward through the symbol tables until a closed scope is encountered, at which point error recovery is invoked, as described later.

When a changed declaration is encountered, the symbol table entry is modified to remove the old definition and insert the new one. The tree comparison routine will then recursively compare deeper in the parse tree below the declaration.

This is the place where the benefit of using keys in the list comparison routines becomes most apparent. If not for the keys, a changed declaration would appear as a deletion and an insertion. While this would be correctly compiled, it would be at great expense, because the declaration may be a procedure with a large enclosed scope. By using the keys, we detect most cases when a scope has changed little.

4.3.3 Updating the Dependency Links

The second phase similarly traverses the parse tree, creating for each symbol table entry a list of its references in the tree. When an identifier that is referenced in the program is seen by this phase, the compiler can quickly determine the corresponding definition by looking it up in the immediately enclosing aggregate or, failing that, in more global aggregates of an open scope. This phase is a straightforward enumeration of the symbol table references. Deleted and inserted references are put on the symbol's reference list.

Special care must be taken to maintain the invariants of the aggregates. When a reference is made to a symbol not in the current context of an open scope, an entry is made and an implicit import is linked to the enclosing scope, recursively as before. When the last definition is removed from an imported identifier, its entry in the aggregate and the dependency link to the enclosing scope is removed.

At the completion of this phase, the symbol table contains all of the dependency links involving attributes in the aggregate.

4.3.4 Evaluating the Dependency graph.

The final phase of semantic analysis evaluates the attributes of the *compound dependency graph*, the union of the attributes from the attribute grammar and those involving the aggregates. This dependency graph may be used to schedule evaluation of the attributes using a parallel variant of the standard topological sort algorithm [Aho *et al.*, 1974].

If each processor is kept busy when there are tasks available, the time required to evaluate this dependence graph is guaranteed to be within a constant factor[6] of an optimal schedule [Graham, 1976]. Thus, the time to perform semantic analysis using this technique is dependent only on the depth of the semantic dependency graph, which is the best we can hope for.

4.3.5 Error detection and recovery

There are three main types of errors that can be detected by the symbol table maintenance phases of semantic analysis. There are multiple definitions, undefined references, and circular definitions. In addition, the symbol table maintenance scheme described here must construct a consistent description of even erroneous programs, to enable future corrections to bring the tables into a consistent state.

To express multiple declarations, each symbol maintains a list of definitions. Although this list will be of length one for correct programs, programs with multiple definitions may be expressed consistently in the symbol table. Thus, the compiler is able to incrementally compile a program whose last version was semantically incorrect.

So that these cases can be efficiently located and reported to the user, a list is maintained at each aggregate (not shown in the pseudocode for clarity) of all multiply defined and undefined symbols, and a count of all errors occuring anywhere within the

[6]The constant is, in fact, two.

scope. The list of errors can be enumerated in linear sequential time[7] to inform the users of these errors. Undefined symbols in an open scope are handled by importing a definition from the parent scope. In a closed scope an unidentified symbol causes a dummy definition to be inserted, and this definition is made to depend on an entry on the list of errors. When a definition is removed from a multiply defined symbol, or a definition is provided for an undefined symbol, or the last reference is removed to an undefined symbol, the entry is removed from the error list. Thus, a program with no multiply-defined and no undefined symbols will have an empty error list.

The final form of error that can be detected by symbol table analysis is circular definitions of attributes. This is detected during scheduling of the evaluation of the attributes, when there remain attributes to evaluate in the graph, yet the ready list is empty and no processors are busy.

4.3.6 Complexity

The data structure used to represent aggregates plays an important role in the complexity of semantic analysis. We require a search structure that uses space linear in the number of elements and allows significant concurrency in the operations. Lock-coupling tree schemes satisfy the linear space requirement, but allow only $O(\log n)$ simultaneous operations. The two-level hash table described in Chapter 2, Figure 2.4.3, allows p processors simultaneously, but its space requirements are linear in the number of processors. The *sibling trie* [Parker, 1989] is suited to this application, requiring only $O(n)$ space and $O(logn)$ time per operation. We will assume the use of this data structure for the representation of search structures in the aggregates.

Symbol table maintenance is limited only by the depth of the parse tree. Since each symbol table operation may have to traverse the symbol tables from the deepest

[7]There is no reason to report errors concurrently, since users can only process then one at a time.

search structure to that at the root of the tree, and using the $O(\log n)$ program depth assumption discussed in the parsing chapter, each operation on a symbol may require a total of $O(\log^2 n)$ time. These may be performed concurrently by $O(n)$ processors, or by $O(n/\log n)$ processors in $O(\log^3 n)$ time. If the program has constant nesting depth of scopes, this becomes $O(\log n)$ time with $O(n)$ processors or $O(\log^2 n)$ time with $O(n/\log n)$. processors.

Evaluating the dependency graph, on the other hand, is limited only by its depth. As we discussed in the previous chapter, the worst schedule requires at most twice the time of an optimal schedule. We have introduced intermediate definitions at every scope between a definition and its use in a nested open scope, and these will contribute at most an additional $O(\log n)$ time (by the nesting depth assumption) to the time required. Therefore, the time to evaluate the dependency graph is $O(\log n)$ or $O(d)$ (proportional to its depth), whichever is more.

For both problems, using incremental techniques reduces the processor requirements from linear (for non-incremental processing) to proportional to the number of the affected trees and attributes.

4.4 Conclusions

We have presented a new extension to the attribute grammar formalism, *upward remote aggregates*, that is suitable for the description of programming languages, and we have shown how this formalism may be used to construct highly parallel incremental compilers. Our analysis shows that, under some mild assumptions, we can perform symbol table analysis in polylogarithmic time with a sublinear number of processors. In addition, we have shown how the use of list attributes may significantly reduce the parse tree depth and may be used to express parallel solutions to many problems arising in semantic analysis.

5 Conclusions

The goal of this thesis was to provide and analyze algorithms that can be used to develop parallel incremental compilers with sublinear parallel time complexity. For all phases of compilation, we have described and analyzed algorithms that exploit fine-grained parallelism; and, under some mild assumptions about the language and program being compiled, we have shown that they run in polylogarithmic time using a sublinear number of processors, thus achieving the goal.

There are two main contributions of this work. First, we describe a new grammar formalism based upon the use of special *list productions* that enable a list of items in the target program to be represented as a balanced tree structure, and extend parallel parsing algorithms to the incremental case. Together with a control structure in which every other processor drops out of the computation at each parsing stage, this allows a more complete and formal analysis of parellel parsing than previous approaches.

The second is a new formalism for expressing the scoping structure of programming languages using *upward remote aggregates*. This formalism allows the compiler writer to express the scoping structure more naturally than previous formalisms, and we present a strategy for efficiently updating the dependency graph when a program changes.

We have taken the tasks of parsing and semantic analysis, each of which is usually expressed as a single phase in a compiler design, and divided them up into a number

of mutually sequential but individually parallel phases. From the point of view of algorithm complexity, we have lost nothing by sacrificing this constant factor. What we have gained is the capability to design much simpler and therefore analytically tractable phases. This should be the first priority in algorithm design for any problem.

5.1 Parallel Algorithm Design

This thesis is also an extended example of parallel program design for a familiar but realistically complex problem. Rather than present the design of a single compiler in detail, we have presented a set of techniques that can be used to construct tools for building parallel incremental compilers for different languages. Therefore, we have relied on techniques that are suitable for describing compilers in formal terms. The two main description languages used were context-free grammars extended with list productions and attribute grammars extended with upward remote aggregates.

We have applied a number of techniques repeatedly in the design. For example, we exploit data structures that display the property of *order independence*: the final state of the data structure is uniquely determined by the set of operations performed upon it, independent of the order in which they were performed. Though this may sound obvious as a design principle, the standard sequential algorithms for these problems are not always order independent, making them difficult to adapt to the parallel case. This is an important property of concurrent data structures, that enable them to be used in an algorithm with little need for synchronization; we have used it throughout the design.

A good example is provided by the implementation of parsers. Conventional LR parsers compute the *canonical* derivation of the input string by sequentially applying productions in order defined by this derivation. This is necessarily a sequential process. But the restriction is unnecessary for the purposes of parsing strings. This is one place

that our design has removed unnecessary sequential constraints from a conventional sequential problem solution; our parsers are *noncanonical* and may perform reductions in any order, subject only to the restriction that a reduction is delayed until all reductions have been performed in descendent nodes of the parse tree. This restriction makes it possible to construct the nodes of the parse tree as the parse proceeds. Furthermore, reductions in nonoverlapping portions of the parse tree may proceed in any order or simultaneously. These generalizations of the parsing process have allowed us to exploit a great deal of parallelism.

Another example is provided by the semantic analysis phase that constructs the symbol tables. Conventional, sequential algorithms for maintaining the symbol table require the definitions to be processed in source order, so that a different referencing environment, which results after each declaration, can be described by modifying the symbol table from the previous environment. In fact, the problem allows much more parallelism. The only ordering constraints are that all declarations affecting an identifier reference be processed before the reference. We have accomplished this simply by processing all of the definitions before all of the references. Thus, a problem that has conventionally been treated as one phase is broken into two, exposing significant parallelism.

Another data structure design technique we have used repeatedly is the *reversible data structure*: data structures whose operations and their inverses are available. Data structures that are reversible help to implement incremental algorithms, where changes to the problem specification can be reflected in the program state by reversing operations on a data structure. This technique was used in semantic analysis for the maintenance of the symbol tables, where we made extensive use of concurrent search structures.

The parser is also a good example of this technique. We consider parsing to be the process of taking a list of symbols, initially containing only terminal symbols (or tokens)

and performing reductions on sublists, replacing the sublist with a nonterminal symbol. When this process terminates, the symbol list has been reduced to the start symbol and the program has been parsed. With this slight reformulation of LR parsing, the *reduce* operation can be undone simply by replacing the nonterminal symbol with the symbols below it in the syntax tree. This basic technique provides the insight as to how parallel parsing has been made incremental.

Symbol table maintenance provides a further example. The data structure representing the aggregate, or symbol table, maintains the interdependencies between symbol definitions; the data structure allows definitions and dependencies to be asserted and retracted in arbitrary order during updates, but the final form of the dependency graph is dependent only on the set of operations that were not retracted.

Finally, applicative data structures are well suited to both parallel and incremental algorithms. By constructing new data structures rather than modifying old ones, the synchronization requirements of mutable shared memory become unnecessary. We have used applicative data structures extensively in syntactic analysis, particularly the data structure *applicative concatenable lists*.

There are some common threads in the algorithm design as well. The primary design principle is the top-down division of the translation task into a number of much simpler mutually sequential translation steps, or phases, each of which is highly parallel. This adds a constant factor to the running time of the algorithm because of the multiple passes over the source program, but it simplifies the subproblems sufficiently that they can be solved in a manner amenable to analysis.

Within the individual phases of the compiler, we have relied heavily on the techniques of parallel divide-and-conquer. Algorithms that use this technique can be designed to exploit parallelism to a very fine grain. In addition, there are programming systems

designed specifically for this class of algorithms that automatically cluster tasks into groups suitable for the host architecture, making divide-and-conquer algorithms efficient for a wide variety of architectures.

5.2 Programming Languages

Historically, the programming language design has been heavily influenced by the technology available for developing compilers. The technologies introduced by this work imply some changes in the constraints imposed upon the language designer. The lexical structure and syntactic structure of the language should have *locality*, that is, the compiler should be able to find its place in the program without reading much of it. For lexical analysis, for example, it helps to limit line length and disallow comments and string constants that cross line boundaries. We have seen how this property can make parallel scanning and parsing much more efficient.

Our techniques for semantic and scope analysis remove from the language designer the constraint that forward referencing should be disallowed. While such restricted languages are easier to implement in sequential compilers, exactly the opposite is true for parallel compilers. In addition, the improved compilation time resulting from the use of parallelism in the compiler may make it practical to reconsider language and compiler features that were previously considered too time consuming.

5.3 Practical Considerations

The *granularity* of parallelism is an important factor affecting the speedup a parallel algorithm can achieve; it is the amount of processing given to a single processor. If the task is to be distributed among a large number of processors, the granularity must necessarily be fine. However, there is usually some communication and synchronization overhead to a parallel implementation that depends on the host machine's architecture,

so there are disadvantages associated with too fine a grain. In addition, the naive strategy of creating a separate process for every potentially parallel task has much too high an overhead to provide a practical implementation. The grain of parallelism should be no finer than necessary to use the available number of processors.

If a very fine granularity is identified, then the grains can usually be grouped dynamically in the implementation to achieve a granularity appropriate for a particular compilation. For this reason, it is generally useful to identify parallelism at the finest grain possible. This is the approach that we have taken in our algorithm design; we place the burden of making this efficient upon the implementor of the parallel programming environment in which the compiler resides. Vandevoorde's `WorkCrews` [1988] is an example of a programming library supporting this paradigm. The scheduling strategy employed by `WorkCrews` also improves the locality of data for computations that are scheduled on the same processor.

5.4 Impact on Programming Environment

The reduced compilation time resulting from a synthesis of parallel and incremental techniques in the programming environment would cause a shorter programming/debugging cycle, and a resulting increase in programmer productivity. Language and compiler features that are now considered too expensive on the basis of increased compilation time may well become practical.

Perhaps the most important impact of this work in the future may be the practical application of fast incremental compilers to a fully integrated environment, in which debugging actions can be expressed in the source language without destroying the program's execution state.

5.4.1 Future Work

We have only begun to scratch the surface of the problems discussed in this thesis. In every direction, there are interesting open problems.

We have described a property of the syntactic structure of programming languages, *syntactic locality*, that enables efficient parallel parsing. We have not explored the formal properties of syntactic locality, which would be useful for the language designer, but we have seen its impact on the complexity of parsing. This, and parse tree depth, limit the number of processors that can be used to parse efficiently.

An implementation of the parser described here would be useful to measure its performance relative to sequential techniques, to measure the interaction of parallel and incremental parsing, and to measure the impact of list productions and various language and grammar features on the parsing efficiency. Is there a simple characterization of locality that allows us to design languages for efficient parallel parsing? An implementation would also help discover which global control strategies improve the efficiency of parsing— for instance, when should the processors drop out of the parsing pipeline? Can these algorithms be extended to LALR(1) or LR(1) formalisms?

There is some overhead associated with dividing programs for parallel parsing; an implementation would help to discover at what point and to what extent the techniques described speed up the compilation process. There is also a question of space efficiency: the compiler is made incremental by saving intermediate results from previous compilations. Many of the algorithms described have added tag fields to nodes of the parse tree. In addition, the parallel version of the compiler constructs large intermediate data structures that are not necessary in the sequential case. It would be nice to know to what extent these techniques increase the memory requirements of a compiler, and what are the limitations to the techniques.

There are a number of target language features that we have not considered, that would increase the difficulty of our task. For instance, some language (such as C) have textual macros that are not lexically scoped that modify the syntax of the language. We conjecture that an incremental macro preprocessor could be designed to output a set of changes from the previous compilation.

When designing any parallel program that uses dynamic data structures, it is convenient to assume the existence of dynamic memory allocation primitives. Their existence provides us with what we need for many of the tasks in code generation, of which memory allocation is a subtask. It is not yet clear from the literature how well the known parallel memory allocation primitives scale. However, a suitable concurrent search structure with a least-upperbound operation would provide us with the means to implement an efficient memory allocator. This is a fundamental and important problem, but too far afield of the thesis topic for a systematic study to be given here.

We have not considered algorithms for parallel code optimization. As a practical matter, production quality compilers require optimization. Unfortunately, with current technology, optimization is incompatible with integrated environments because the transformation may obscure debugging information and the mapping between the running program and its source representation. Our algorithms, however, can be used in the non-incremental case for parallel compilation, and there optimization is a much more important problem.

Bibliography

[Aho *et al.*, 1974] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[Allen, 1987] Stanley Allen, "Parallelization of Compiler Phases on a Multiprocessor Machine," Technical Report TAMU-87-009, Texas A&M University, May 1987.

[Andre *et al.*, 1981]. F. Andre, J. P. Banatre, and J. P. Routeau, "A Multiprocessing Approach to Compile-Time Symbol Resolution," *ACM Transactions on Programming Languages and Systems*, 3(1):11–23, January 1981.

[Baer and Ellis, 1977] J. L. Baer and C. Ellis, "Model, Design, and Evaluation of a Compiler for a Parallel Processing Environment," *IEEE Transactions on Software Engineering*, 3(6):394–405, November 1977.

[Banatre *et al.*, 1979] J. P. Banatre, J. P. Routeau, and L. Trilling, "An Event-Driven Compiling Technique," *Communications of the ACM*, 22(1):34–42, January 1979.

[Beshers and Campbell, 1985] George McArthur Beshers and Roy Harold Campbell, "Maintained and Constructor Attributes," In *Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments*, pages 34–42, July 1985.

[Boehm and Zwaenopoel, 1987] Hans-Juergen Boehm and Willy Zwaenopoel, "Parallel Attribute Grammar Evaluation," Technical Report COMP TR87-55, Rice University, June 1987.

[Christopher *et al.*, 1981] T. W. Christopher, O. I. El-Dessouki, M. W. Evens, H. Harr, H. Klawans, P. Krystosek, R. Mirchandani, and Y. Tarhan, "SALAD - A Distributed Compiler for Distributed Systems," In *Proceedings of the 1981 IEEE International Conference on Parallel Processing*, August 1981.

[Cohen *et al.*, 1982] Jacques Cohen, Timothy Hickey, and Joel Katcoff, "Upper Bounds for Speedup in Parallel Parsing," *Journal of the ACM*, 29(2):408–428, April 1982.

[Cohen and Kolonder, 1985] Jacques Cohen and Stuart Kolonder, "Estimating the Speedup in Parallel Parsing," *IEEE Transactions on Software Engineering*, 11, 1:114–124, January 1985.

[Delisle *et al.*, 1984] Norman M. Delisle, David E. Menicosy, and Mayer D. Schwartz, "Viewing a Programming Environment as a Single Tool," *ACM SIGPLAN Notices*, 19(5):49–56, May 1984.

[Ellis, 1985] Carla Schlatter Ellis, "Concurrency and Linear Hashing," Technical report, University of Rochester, March 1985.

[Feiler, 1982] Peter Hermann Feiler, *A Language-Oriented Interactive Environment Based on Compilation Technology*, PhD thesis, Carnegie-Mellon University, May 1982.

[Feldman, 1979] Stuart I. Feldman, "Make—A Program for Maintaining Computer Programs," *Software—Practice and Experience*, 9:255–265, 1979.

[Fischer *et al.*, 1984] C. N. Fischer, Gregory F. Johnson, Jon Mauney, Anil Pal, and Daniel L. Stock, "The Poe Language-Based Editor Project," In *Proceedings of the*

SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pages 21–29, May 1984.

[Fischer, 1975] Charles N. Fischer, *On Parsing Context-Free Languages in Parallel Environments*, PhD thesis, Cornell University, April 1975.

[Fritzson, 1982] Peter Fritzson, "Fine-Grained Incremental Compilation for Pascal-Like Languages," Technical report, University of Linkoping, July 1982.

[Fritzson, 1983] Peter Fritzson, "Symbolic Debugging Through Incremental Compilation in an Integrated Environment," *The Journal of Systems and Software*, 3:285–294, 1983.

[Fritzson, 1984] Peter Fritzson, "Preliminary Experience from the DICE system - a Distributed Incremental Compiling Environment," *ACM SIGPLAN Notices*, 19(5):113–123, May 1984.

[Garey and Johnson, 1979] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

[Garrison, 1987] Phillip Edward Garrison, *Modeling and Implementation of Visibilty in Programming Languages*, PhD thesis, University of California at Berkeley, December 1987.

[Ghezzi and Mandriolli, 1979] Carlo Ghezzi and Dino Mandriolli, "Incremental Parsing," *ACM Transactions on Programming Languages and Systems*, 1(1), July 1979.

[Graham, 1976] R. L. Graham, "Bounds on the Performance of Scheduling Algorithms," In E. G. Coffman, Jr., editor, *Computer and Job-Shop Scheduling Theory*, chapter 5, pages 165–227. John Wiley and Sons, New York, 1976.

[Graham and Rhodes, 1975] Susan L. Graham and Steven P. Rhodes, "Practical Syntactic Error Recovery," *Communications of the ACM*, 18(11), November 1975.

[Halstead, 1985] Robert Halstead, "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems*, October 1985.

[Hoover, 1987] Roger Hoover, *Incremental Graph Evaluation*, PhD thesis, Cornell University, May 1987.

[Hoover and Teitelbaum, 1986] Roger Hoover and Tim Teitelbaum, "Efficient Incremental Evaluation of Aggregate Values in Attribute Grammars," In *Proceedings of the ACM SIGPLAN 1986 Symposium on Compiler Construction*, pages 39–50, July 1986.

[Horgan and Moore, 1984] J. R. Horgan and D. J. Moore, "Techniques for Improving Language-Based Editors," In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM, May 1984.

[Huen *et al.*, 1977] W. Huen, O. El-Dessouki, E. Huske, and M. Evans, "A Pipelined DYNAMO Compiler," In *Proceedings of the 1977 IEEE International Conference on Parallel Processing*, pages 57–66, August 1977.

[Jalili and Gallier, 1982] Fahimeh Jalili and Jean H. Gallier, "Building Friendly Parsers," In *SIGPLAN Symposium*. ACM, June 1982.

[Johnson, 1984] Gregory F. Johnson, *An Approach to Incremental Semantics*, PhD thesis, University of Wisconsin at Madison, July 1984.

[Johnson and Fischer, 1982] Gregory F. Johnson and Charles N. Fischer, "Non-syntactic Attribute Flow in Language Based Editors," In *Conference Record of the*

Ninth Annual ACM Symposium on Principles of Programming Languages, pages 185–195, 1982.

[Kaplan and Kaiser, 1986] Simon M. Kaplan and Gail E. Kaiser, "Incremental Attribute Evaluation in Distributed Language-Based Environments," In *Proceedings of the ACM Principles of Distributed Computing*, pages 121–130, 1986.

[Karr, 1984] Michael Karr, *Code Generation by Coagulation*, PhD thesis, Harvard University, May 1984.

[Khanna *et al.*, 1990] Sanjay Khanna, Arif Ghafoor, and Amrit Goel, "A Parallel Compilation Technique based on Grammar Partitioning," In *Proceedings of the ACM Annual Conference*, February 1990.

[Klein and Reif, 1988] Philip N. Klein and John H. Reif, "Parallel Time O(log n) Acceptance of Deterministic CFLs on an Exclusive-Write P-RAM," *SIAM Journal on Computing*, 17(3):463–485, June 1988.

[Ladner and Fischer, 1980] Richard E. Ladner and Michael J. Fischer, "Parallel Prefix Computation," *Journal of the Association for Computing Machinery*, 27(4):831–838, October 1980.

[Ligett *et al.*, 1982] Dan Ligett, Glen McCluskey, and W. M. McKeeman, "Parallel LR Parsing," Technical Report TR-82-03, Wang Institute of Graduate Studies School of Information Technology, July 1982.

[Lipkie, 1979] D. Lipkie, *A Compiler Design for Multiple Independent Processor Computers*, PhD thesis, University of Washington, 1979.

[Low, 1988] James R. Low, "Lexical Analysis on a Moderately Sized Multiprocessor," Technical Report TR261, University of Rochester, October 1988.

[Mickunas and Schell, 1978] M. D. Mickunas and R. M. Schell, "Parallel Compilation in a Multiprocessor Environment," In *Proceedings of the ACM Annual Conference*, pages 241–246, 1978.

[Miller and LeBlanc, 1982] J. A. Miller and R. J. LeBlanc, "Distributed Compilation: A Case Study," Technical report, Georgia Institute of Technology, 1982.

[Morris and Schwartz, 1981] Joseph M. Morris and Mayer D. Schwartz, "The Design of a Language-Directed Editor for Block Structured Editors," *ACM SIGPLAN*, 10, 6, June 1981.

[Myers, 1984] Eugene W. Myers, "Efficient Applicative Data Types," In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, 1984.

[Park and Burnett, 1979] William T. Park and David J. Burnett, "An Interactive Incremental Compiler for more Productive Programming of Computer-Controlled Industrial Robots and Flexible Automation Systems," Technical report, SRI International, February 1979.

[Parker, 1989] Jeff D. Parker, "A Concurrnet Search Structure," *Journal of Parallel and Distributed Computing*, 7:256–278, 1989.

[Pennello and DeRemer, 1978] Thomas J. Pennello and Frank DeRemer, "A Forward Move Algorithm for LR Error Recovery," In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, January 1978.

[Reps *et al.*, 1986] T. Reps, C. Marceau, and T. Teitelbaum, "Remote Attribute Updating for Language-Based Editors," In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 1–13, January 1986.

[Reps, 1982] Thomas Reps, *Generating Language-Based Environments*, PhD thesis, Cornell University, August 1982.

[Sarkar and Deo, 1986] Dilip Sarkar and Narsingh Deo, "Estimating the Speedup in Parallel Parsing," In *Proceedings of the 1986 IEEE International Conference on Parallel Processing*, pages 157–163. IEEE, August 1986.

[Schell, 1979] Richard Marion Schell, Jr., *Methods for Constructing Parallel Compilers for use in a Multiprocessor Environment*, PhD thesis, University of Illinois at Urbana-Champaign, 1979.

[Schwartz *et al.*, 1984] Mayer D. Schwartz, Norman M. Delisle, and Vimal S. Begwami, "Incremental Compilation in Magpie," In *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction*, pages 122–131, June 1984.

[Seshadri *et al.*, 1988] V. Seshadri, D. B. Wortman, M. D. Junkin, S. Weber, C. P. Yu, and I. Small, "Semantic Analysis in a Concurrent Compiler," In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 233–239. ACM SIGPLAN, 1988.

[Seshadri *et al.*, 1987] Venkatadri Seshadri, Ian S. Small, and David B. Wortman, "Concurrent Compilation," In M. H. Barton, E. L. Dagless, and G. L. Reijns, editors, *Proceedings of the IFIP WG10.3 Working Conference on Distributed Processing*, pages 627–641. IFIP, Elsevier Science Pubishers B. V. (Horth-Holland), October 1987.

[Shasha and Goodman, 1988] Dennis Shasha and Nathan Goodman, "Concurrent Search Structure Algorithms," *ACM Transactions on Database Systems*, 13(1):53–90, March 1988.

[Skillicorn and Barnard, 1988] D. B. Skillicorn and D. T. Barnard, "Parallel Parsing on the Connection Machine," Technical Report TR 88-209, Queen's University, January 27 1988.

[Stromberg, 1982] Dan Stromberg, "Text Editing and Incremental Compiling," Technical Report LiTH-MAT-R-82-34, University of Linkoping, Sweden, October 1982.

[Teitelbaum *et al.*, 1981] Tim Teitelbaum, Thomas Reps, and Susan Horowitz, "The Why and Wherefore of the Cornell Program Synthesizer," In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 8–16, June 1981.

[Ullman, 1976] J. D. Ullman, "Complexity of Sequencing Problems," In E. G. Coffman, Jr., editor, *Computer and Job-Shop Scheduling Theory*, chapter 4, pages 139–164. John Wiley and Sons, New York, 1976.

[Vandevoorde and Roberts, 1988] Mark T. Vandevoorde and Eric S. Roberts, "WorkCrews: An Abstraction for Controlling Parallelism," *International Journal of Parallel Programming*, 17(4), 1988.

[Vandevoorde, 1988] Mark Thierry Vandevoorde, "Parallel Compilation on a Tightly Coupled Multiprocessor," Technical report, Digital Systems Research Center, March 1 1988.

[Waters, 1982] Richard C. Waters, "Program Editors Should Not Abandon Text Oriented Commands," *ACM SIGPLAN Notices*, 17(7), July 1982.

[Wirth, 1982] Niklaus Wirth, *Programming in Modula-2*, Springer-Verlag, 1982.

[Yu, 1989] Chun Pong Yu, "Practical Parallel Lexing," Technical Report CSRI-226, University of Toronto, May 1989.

[Zosel, 1973] M. Zosel, "A Parallel Approach to Compilation," In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 59–70, 1973.